НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО”

Факультет Інформатики та обчислювальної техніки
(повне найменування інституту, факультету)

Обчислювальної техніки
(повна назва кафедри)

До захисту допущено
Завідувач кафедри
_____ *Сергій СТІРЕНКО*
(підпис)

“___”_____2020р.

Дипломний проект

на здобуття ступеня бакалавра

з напряму підготовки 6.050102 « Комп’ютерна інженерія »

на тему: «Чат на основі веб-сервісу»

Виконав: студент __4__ курсу, групи __ІО - 61_____

Коваленко Антон Сергійович _____ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник старший викладач Виноградов Юрій Миколайович _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант нормо контроль д.т.н., проф.Сімоненко В.П. ___ _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки
(повне найменування інституту, факультету

Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень **бакалавр**

Напрям підготовки **6.050102 « Комп’ютерна інженерія »**

ЗАТВЕРДЖУЮ

Завідувач кафедри

Сергій СТІРЕНКО _____
(підпис)

“___”_____20___ р.

ЗАВДАННЯ

на бакалаврський дипломний проект студента

Коваленко Антону Сергійовичу

1. Тема проекту (роботи) *Чат на основі веб-сервісу*

керівник проекту (роботи) Винградов Юрій Миколайович, затверджені наказом по університету від « *07* » *травня* 2020 р. № 1081-С

2. Термін здачі студентом закінченого проекту (роботи)_____ *2020*р.
3. Вихідні дані до проекту (роботи) *технічна документація, технічне завдання.*
4. Зміст розрахунково-пояснювальної записки: *опис предметної області, дослідження методики побудови електронного підручника на базі гіпертекстової технології, програма забезпечення гіпертекстових технологій.*

5. Консультанта проекту (робота), з вказівкою розділів роботи, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Сімоненко В.П.		

6. Дата видачі завдання___01.09.2019 року___

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів дипломного проекту (роботи)	Строк виконання етапів проекту(роботи)	Примітки
1.	*Затвердження теми роботи*	01.09.2019	
2.	*Вивчення та аналіз завдання*	15.09.2019	
3.	*Розробка архітектури та загальної структури систем*	04.10.2019	
4.	*Розробка структур окремих підсистем*	13.12.2019	
5.	*Програмна реалізація системи*	03.02.2020	
6.	*Оформлення пояснювальної записки*	04.04.2020	
7.	*Передзахист*	26.05.2020	
8.	*Захист*	25.06.2020	

Студент-дипломник _____

(підпис)

Керівник роботи _____

(підпис)

Анотація

В даній дипломній роботі були проаналізовані існуючі архітектури та підходи в створенні онлайн-чатів. Було запропоновано архітектуру веб-чату, в якій було усунуто, або мінімізовно всі недоліки існуючих реалізацій. Було розроблено високонавантажену систему на прикладі веб-чату. Архітектуру чату було створено так, що її лекго інтегрувати з іншими системами, або побудувати на її основі більш складну систему.

Annotation

In this research the existing architectures and approaches towards creating online chats were analysed. The suggested web-chat architecture does not contain the flaws of all current realisations as they were eliminated and minimised. The highload system based on the web chat was created. Chats architecture was established to be easily integrated in existing systems.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЕКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1.	А4		Завдання на дипломний проект	2	
2.	А4	ІАЛЦ.467100.002 ТЗ	Технічне завдання	3	
3.	А4	ІАЛЦ.467100.003 ПЗ	Пояснювальна записка	60	
4.	А4	ІАЛЦ.467100.004 Д1	Чат на основі веб-сервісу. Схема структурна.	1	
5.	А4	ІАЛЦ.467100.005 Д2	Чат на основі веб-сервісу. Діаграма станів.	1	
6.	А4	ІАЛЦ.467100.006 Д3	Чат на основі веб-сервісу. Схема функціональна	1	
7.			Чат на основі веб-сервісу. Лістинг програми	16	

ТЕХНІЧНЕ ЗАВДАННЯ

до дипломної роботи
Освітньо-кваліфікаціййного рівні бакалавр

на тему: «Чат на основі веб-сервісу»

Київ – 2020

ЗМІСТ

					ІАЛЦ.467100.002 ТЗ		
Зм	*Арк.*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		*Коваленко А.С.*			Чат на основі веб-сервісу. *Технічне завдання.*	*Літ.*	*Арк.* 1 / *Аркушів* 4
Перевір.		*Виноградов*					
Н.Контр.		*Сімоненко*					
Затверд.							

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку курсу «Інженерія програмного забезпечення». Область застосування: практичне використання людьми при користуванні комп’ютерними засобами.

2. ПІДСТАВИ ДЛЯ РОЗРОКИ

Підставою для розробки є завдання на виконання роботи кваліфікаційно-освітнього рівня «бакалавр комп'ютерної інженерії», затверджене кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проекту є розробка високонавантаженої системи на прикладі веб-чату. Призначення даного проекту є дати можливість необмеженій кількості користувачів користуватись чатом.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література з теорії і практики програмування, бакалаврські роботи інших студентів, публікації в Інтернеті з даних питань.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробляємого продукту

• Розроблена архітектура повинна бути створена згідно з стандартам SDN

• Розроблена архітектура повинна бути гнучкою та дозволяти горизонтальне масштабування системи.

• Система повинна бути доволі швидкою аби не створювати користувачеві незручностей.

5.2. Вимоги до програмного забезпечення

• Linux-подібна операційна система.

• Наявність на комп’ютері Node.js не нижче версії 10.

5.3. Вимоги до апаратної частини

• Оперативної пам’яті не менше 512 Мбайт.

• Вільне місце на жорсткому диску не менше 500 Мбайт.

• Процессор 1.4 ГГц

6. ЕТАПИ РОЗРОБКИ

	Дата
Вивчення літератури	28.03.2020
Складання і узгодження технічного завдання	03.04.2020
Створення модулів системи, що розробляється	15.04.2020
Тестування окремих модулів системи	25.04.2020
Допрацювання, налагодження і виправлення помилок	01.05.2020
Оформлення документації дипломної роботи	15.05.2020

Пояснювальна записка
до дипломного проекту

на тему: «Чат на основі веб-сервісу»

Київ – 2020

ЗМІСТ

ВСТУП

Одним з найефективніших на сьогодні способів комунікувати є чати та месенджери. Це твердження легко підтверджується великою популярністю таких сервісів. Таке явище доволі легко пояснити, адже такі сервіси доволі зручні та швидкі, всі ми знаємо що написати декілька повідомлень завжди легше аніж користуватись стільниковим зв'язком.

В часи зародження інтернету, такі сервіси були доволі примітивними і повільними. Користувачі не могли відслідкувати такі звичайні для нас сьогодні речі, як «статус онлайн» і кількість непрочитаних повідомлень, а повільна робота сервісу нікого не дивувала. Але з часом, як і будь-які популярні інтернет-сервіси, чати набували складності та функціональності.

З великою популярністю, відповідно, ростуть і вимоги до таких сервісів. І так на сьогодні головною складністю такого роду сервісів є стійкість до величезного навантаження, яке продукується мільйонами користувачів. Коли десять років назад користувачі з легкістю «пробачали» сервісу повільну роботу, то на сьогоднішній день, про схожі компроміси мова йти не може, повільна система прирівнюється до взагалі неробочої.

Також одним з найважливіших «завдань» чату є забезпечення безпеки користувачів, оскільки, на сьогоднішній день, хоча б декілька випадків «взлому» популярного веб-сервісу можуть назавжди знищити його, адже друге чим не можуть поступитись користувачі після швидкодії – це персональні дані. Не дивлячись на сучасні технології, більшість сучасних веб-чатів дуже часто мають вище перелічені проблеми та недоліки, що є неприпустимо. До того ж, в епоху диджиталізації, коли месенджерами, для зручності, швидкості та ефективності користуються державні інституції та бізнес-структури, питання захисту персональних даних та конфіденційності стають пріоритетними.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ТЕХНОЛОГІЙ В СТВОРЕННІ ВЕБ-ЧАТІВ

1.1 Аналіз недоліків існуючих систем

Серверні системи обмеженої пропускної здатності. Найчастіше в створенні онлайн-чатів використовують один сервер, який являє собою одну обчислювану машину (далі машина-хост), яка обмежена в потужності, тому така система неспроможна витримувати велике навантаження. Коли розробники мають справу з даною проблемою вони намагаються збільшити потужність системи.

Один з варіантів збільшити пропускну здатність сервера - збільшувати потужніть ‘машини-хоста’, але такий спосіб дуже обмежений, адже дуже швидко можливість нарощувати потужність зникає в силу сьогоднішніх технологій.

Є інший спосіб покращити пропускну здатність - горизонтальне масштабування, але й тут є свої проблеми - оскільки спочатку система будувалась як монолітний сервер (сервер, що запускається на одній машині), то архітектуру системи дуже складно перебудувати для того щоб запустити її на багатьох машинах.

Присутні й інші проблеми, такі як втрата данних. Оскільки сучасні системи часто використовують бази данних, які розміщують свої данні в оперативній пам’яті, часто процес такої БД зупиняється і данні втрачаються. Щоб запобігти таким проблемам, використовується реплікація данних в постійне сховище данних, наприклад на жорсткий диск, але часто така реплікація налаштована неправильно і данні все ж втрачаються.

В багатьох системах, як протокол передачі данних, використовується http протокол, який суттєво поступається rpc протоколу в швидкодії, гнучкості та зручності.

1.2 Аналіз підходів побудови API сервера

1.2.1. REST

REST (Representational state transfer) – це стиль архітектури пррограмного забеспечення для розподілених сисем, які як правило використовуються для побудови веб-сервісів та веб-служб[1].

Кожна одиниця інформації визначає URL, наприклад 5 повідомлення буде представлятись таким чином - /messages/5.

Управляння інформацією сервіса відбувається за рахунок протоколу передачі данних, зазвичай для REST використовують http протокол. Для HTTP дії над данними задаються за допомогою спеціального заголовку - method, GET (отримати), PUT (замінити), PATCH(змінити), POST (створити), DELETE (видалити). Таким чином дії CRUD (Create-Read-Updtae-Delete) можуть виконуватися з всіма 4-ма методами. Також для обробки результатів використовується статус-коди, наприклад 200 (Ok), 404 (Not found), 500 (Internal server error)

На приладі:

GET /messages/ - отримати всі повідомлення

PUT /message/5 - змінити 5 повідомлення

POST /message – додати нове повідомлення

DELETE /message/5 – видалити 5 повідомлення

1.2.2. JSON RPC

JSON RPC (Remote Procedure Call) підхід. Якщо в REST API використовується HTTP протокол, то в RPC використовується Websocket протокол, який може працювати як поверху TCP так і UDP. Данний протокол працює без стоврення постійного з”єднаня, тому сервер ‘сплікується’ з клієнтом в вигляді пересилання подій. Використовується єдиний URL, який і являє собою вхідну точку точку в API.

Сервер приймає запити виду - {"jsonrpc": "2.0", "method": “message.lsend, "params": {“text: “hello world”}, "id": 1}

Повертає відповіді типу - {"jsonrpc": "2.0", "result": {“status”: “ok”}, "id": 1}. Якшо стається помилка сервер повертає - {"jsonrpc": "2.0", "error": {"code": 404, "message": “Not found”:}, "id": “1”}.

Використовуючи RPC протокол ми отримуємо такі переваги - можна не ховати складні операції за набором HTTP-методів та надлишковими URI[2]. Є предметні області, де операцій в API має бути більше ніж сутностей, наприклад проекти з непростими бізнес-процесами, gamedev, месенджери і подібні realtime-системи.

Друга перевага над REST API полягає в тому, що інформація про запит розташована тільки в тілі запиту, в REST інформація може бути - в тілі запиту, в URL, в HTTP-заголовках, тому це дає певну невизначеність, що не є добре[2].

1.3 Аналіз монолітної архітектури сервера

Монолітною архітектурою називається підхід побудови сервера, в якому серверне ПЗ виконується виключно на одній машині та звертається до однієї бази данних.

1.3.1 Переваги підходу

Великою перевагою монолітної архітектури є те, що її легше реалізувати. У монолітній архітектурі розробник можете швидко почати реалізовувати систему, замість того щоб витрачати час на роздуми про взаємодії між процесами.

Монолітна архітектура найкраще підходить для систем з невеликим навантаженням, адже така архітектура дозволяє найшвидше створити систему та спричиняє найменше проблем в майбутній її підтримці[3]. При монолітній архітектурі простіше підтримувати узгодженість коду, обробляти помилки.

Значною перевагою є надійність такої системи, екземпляр серверного застосунку завжди запускається на одній машині-хості, тому будь-яка

взаємодія з програмними компонентами відбувається в межах цієї машини, що виключає цілу низку проблем з передачею данних[3].

1.3.2 Недоліки підходу

Головним недоліком монолітної архітектури вважається обмежене масштабування даної системи, адже в системи немає змоги роз'єднати програмні модулі і винести їх на окремі сервери.

У моноліті практично немає ізоляції. Проблема або помилка в модулі може уповільнити або зруйнувати всю систему. Значним недоліком можна вважати складність введення змін в таку систему, оскільки всі програмні тісно пов'язані - зміна одного модуля часто призводить до неочікуваної зміни інших модулів, що як наслідок призводить до проблем в роботі системи.

1.3.3 Горизонтальне масштабування

Для того щоб збільшити потужність всієї системи, потрібно розмістити еземпляр серверного застосунку на багатьох машинах-хостах, також потрібно налаштувати load-balncer, так, аби він розділяв навантажання на всі сервери[4]. Єдиним «вузьким» місцем такої системи стає load-balncer, адже він пропускає через себе все навантаження. Рішенням такої проблеми може стати збільшення кількості load-balncer, тобто побудови своєрідного кластера. Структура такої системи показана на рис 1.1

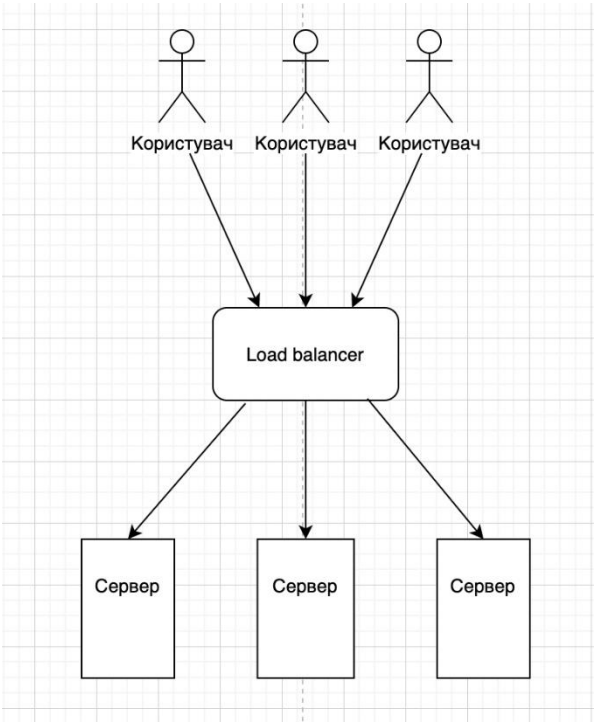


Рисунок №1.1 Структура системи з багатьма екземплярами сервера та Load-balancer

Для того аби горизонатальне мастштабування було можливим, потрібно аби сервер не зберігав в оперативній пам’яті данних, які можуть бути потрібні всім серверам. В разі, якшо така проблема присутня, потрібно перенести роботу з такими данними в спеціальні центрадізовані сховища данних, наприклад Redis, Memcached.

Існує й інша проблема. Якшо сервер виступає в ролі посередника в передачі данних між двома клієнтами (веб-чат), потрібно аби ці клієнти підключились до одного сервера, або сервери використовували спеціальні системи поширення данних, наприклад Redis, які будуть передавати даннів від одного клієнта до іншого.

Головною перевагою такого мастштабування є те, що кількість серверів необмежена, тобто ми можемо мастштабуватись до будь-яких потужностей[4]. При використатнні спеціальних систем, які будь відслідковувати навантаження на сервери, можливо налаштувати спеціальний механізм, який буде запускати і зупиняти сервери в залежності від вхідного навантаження. При зупиненні сервера таким мехінізмом

потрібно обробляти данні, які збережені в оперативній пам”яті сервера, адже після зупинки ми їх втратимо. Така система дозволяє раціонально використовувати серверні потужності та суттєво екномити ресурси.

1.4 Аналіз мікросерверної архітектури сервера

Мікросервісна архітектура це - принципова організація розподіленої системи на основі мікросервісів та їх взаємодії один з одним і з середовищем по мережі, а також принципів, що визначають проектування архітектури та її створення[5]. Оскільки всі мікросерсервіси знаходяться на різних фізичних адресах, для того аби доступ до такого API був відкритий через одну фізичну адресу, в систему вводять спеціальний сервер, який зазвичай називають API-service, який приймає на вхід всі запити до API і пересилає їх до потрібних мікросервісів. Часто таких “внутрішніх” запитів може бути декілька, в такому разі API-servic компонує відповіді мікросервісів перед тим як відіслати результат клієнту. Передача данних між мікросервісами здійснюється механізмами які описані в пункті 1.4.4. Структура такої архітектури зображена на рис 1.2

Відомо, що найбільші гравці на ринку веб-технологій використовують саме мікросервісну архітектуру через її змогу витримувати велике навантаження.

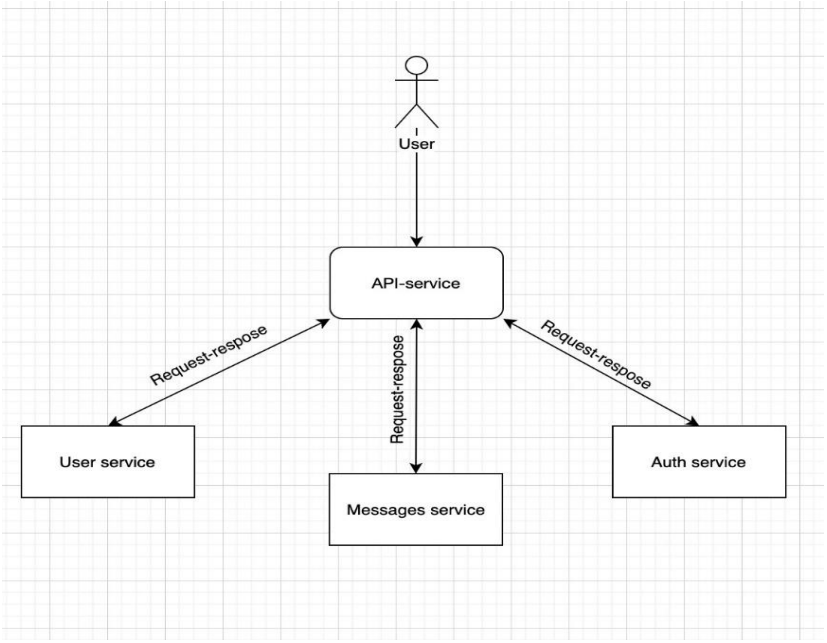


Рисунок 1.2 Структура мікросервісної архітектури

1.4.1 Переваги підходу

Найбільшою перевагою такого підходу вважають велику пропускну здатність системи, адже в основі системи лежить багато серверів, які рівномірно забирають навантаження на себе. Найчастіше саме велика пропускна здатність і є вирішальним фактором, який і змушує розробників будувати систему на основі мікросерверної архітектури.

Великою перевагою мікросервісів є гнучкість в їх розробці, оскільки кожен мікросервіс являє собою окремий сервер, кожен такий сервер може бути розроблений на різних технологіях та різними командами програмістів, яким залишається тільки узгодити інтерфейси вхаємодії між мікросервісами[6].

Великою перевагою є те, що якщо потрібно оновити один з мікросервісів у вже працюючому середовищі, нам не потрібно зупиняти всю систему, а достатньо призупинити мікросервіс який буде оновлятись[6]. При такому оновленні, більшість системи буде працювати і обробляти клієнтські запити

1.4.2 Недоліки підходу

Основиним недоліком такої архітектури є її велика складність. Саме проблема складності часто приводить до того, що система проектується неправильно і як результат – швидкодія нижча ніж в звичайного монолітного сервера. Для того аби побудувати правильні мікросервіси та взаємодію між ними потрібно мати дуже велику кваліфікацію. Для побудови правильної взамодії мікросервісів потрібно дуже добре розбиратись в протоколах передачі данних, в основі яких лежать TCP, UDP, потртрібно добре розумітись в теорії систем масового обслуговування, теорії черг та інших фндаментальних знань в комп'ютерних науках.

Також суттєвою є проблема складності підтримки інфраструктури такої системи, оскільки дуже часто кількість мікросервісів може досягати десятків, прослідкувати за ними стає дуже складно. Також таку систему неможливо запусти локально (на персональному комп'ютері розробника), що дуже ускладнює розробку системи.

1.4.3 Архітектура мікросервіса

Мікросервіс - це відокремлене серверне прграмне забезпечення, що найчатсіше запускається на оркемому сервері, та являє собою ізольований, отомарний програмний модуль[5].

Однією з особливостей мікросервіса є його розмір, мікросервіс повинен бути невеликим, тобто не охоплювати відразу декілька контекстів і бути побудованим навколо однієї бізнес-потреби. Мікросервіс повинен бути максимально ізольваним від інших мікросервісів та спілкуватись з ними тільки за допомогою спеціальних програмних інтерфейсів[6]. Мікросервіс має бути побудованим так, що його зміна чи заміна повина мінімально вплинути на інші мікросервіси, для цього перед побудовою мікросервісів потрібно продумати і розробити архітектуру всієї системи. Також задля кращої ізоляції кожен мікросервіс повинен використовувати окрему базу

данних, оскільки в протилежному випадку, логіка всіх мікросервсів тісно переплiталася, що є неприпустимо в мікросервісній архітектурі.

Зазвичай в складних системах багато мікросервісів, тому для створення нового мікросервісу часто використовують шаблон, який являє собою базову архітектуру мікросервіса, в якому вже реалізована структура файлів, розбиття на логічні модулі…. Такі шаблони використовуються задля зручнішого і швидшого стоврення нового мікросервіса, також в такого підходу є й інша перевага - це зручність підтримки таких мікросервісів, оскільки всі мікросервіси побудовані на основі однакової архітектури, програміст не втрачає час на дослідження структури кожного мікросервісу[6].

1.4.4 Механізми взаємодії мікросервісів

В мікросервісній архітектурі завжди потрібно передавати данні між мікросервісами. Зазвичай запити йдуть від API service до інших мікросервісів, так як показано на схемі Механізми передачі данних можна поділити на такі групи:

- Передача по http;
- Передача по udp;
- Передача данних за рахунок спільного сховища данних.

На вході в систему(API service) обов’язково має бути реалізована механізм обслуговування черги запитів, адже пропускна здатність API-service та інших мікросервісів може відрізнятись, крім того, в моменти великого навантаження система не зможе обробити всі запити що приходять, тому їх потрібно ставити в чергу. Струкрута такої системи показана на рис. 1.3

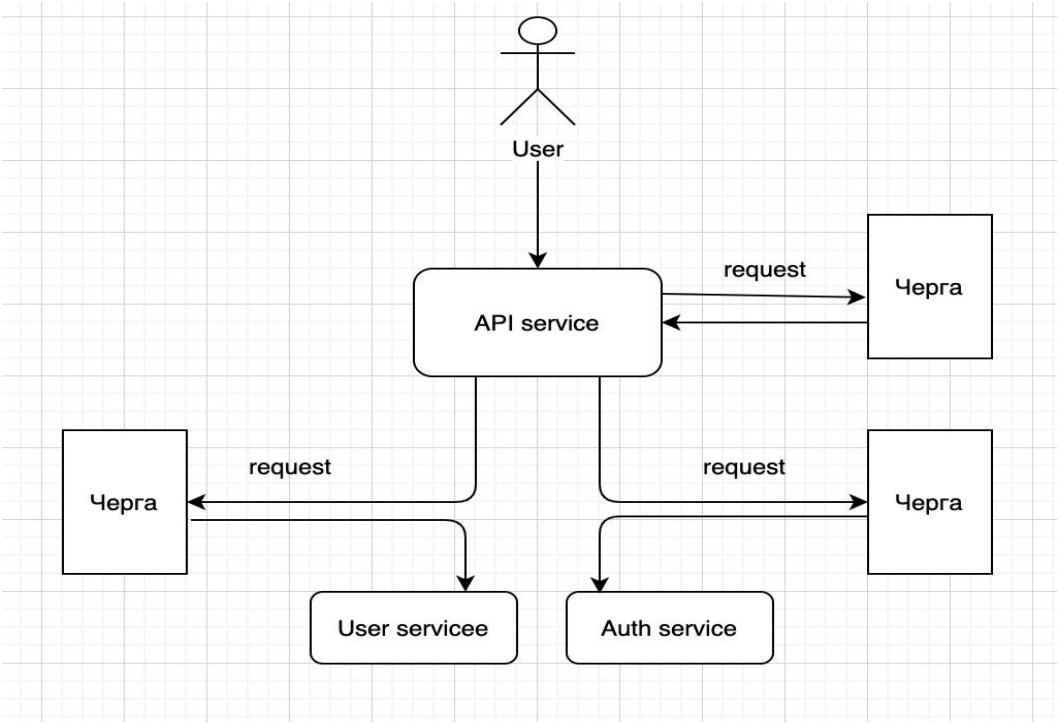


Рисунок 1.3 Спосіб організації черг запитів в мікросервісній архітектурі

Найчастіше механізм передачі даних реалізовують за допомогою спільного сховища данних. В мікросервісну архітекрутуру вводиться надшвидка база данних, яка працює в оперативній пам’яті наприклад Redis. Наприклад, якщо потрбно пересилати данні з API service в User service , в Redis стоворюється спеціальний масив запитів, в який API service буде додавати запити, а User service буде робити запити в Redis на перевірку масива з запитами з певною частотою, наприклад 200мс. Якшо User service знаходить там новий запит, він вичитує його, а на його місце записує результат операії. Також в такій черзі потрібно реалізувати час актуальності кожного запиту, адже кожен запит має крайній час виконання на клієнті, наприклад в браузері (максимально 30 секунд). Redis дозволяю вирішити цю проблему дуже просто – в його ядрі вже реалізвний схожий мехінзм, тобто кожному запису можлна простaвити час його актуальності, після перебігу якого, такий запис буде видалено.

Існує є інший підхід – це організація передачі по UDP протоколу, який є доволі швидкий, але не гарантує доставку данних на відміну від TCP. Але при використанні RPC протоколу, всі данні будуть гартовано доставлені.

Також однією з альтернатив є використання http протоколу. В основі кожного мікросервіса буде створено http сервер, який буде відповідати на запии інших мікросервісів. Такий підхід є застарілим, адже він дуже повільний через http протокол.

1.4.5 Горизонтальне масштабування

Одним з способів є розмноження всієї мікросервісної системи. Для того аби розподілити навантаження по такій системі, встановлюється Load balancer, який буде працювати за алгоритмом round-robin, тобто розподіляти запити послідовно та по колу, наприлад: перший запит піде на першу підсистему, дргий запит на другу підсистему, третій знову на першу і так далі[7]. Структура такої системи показана на рис. 1.4

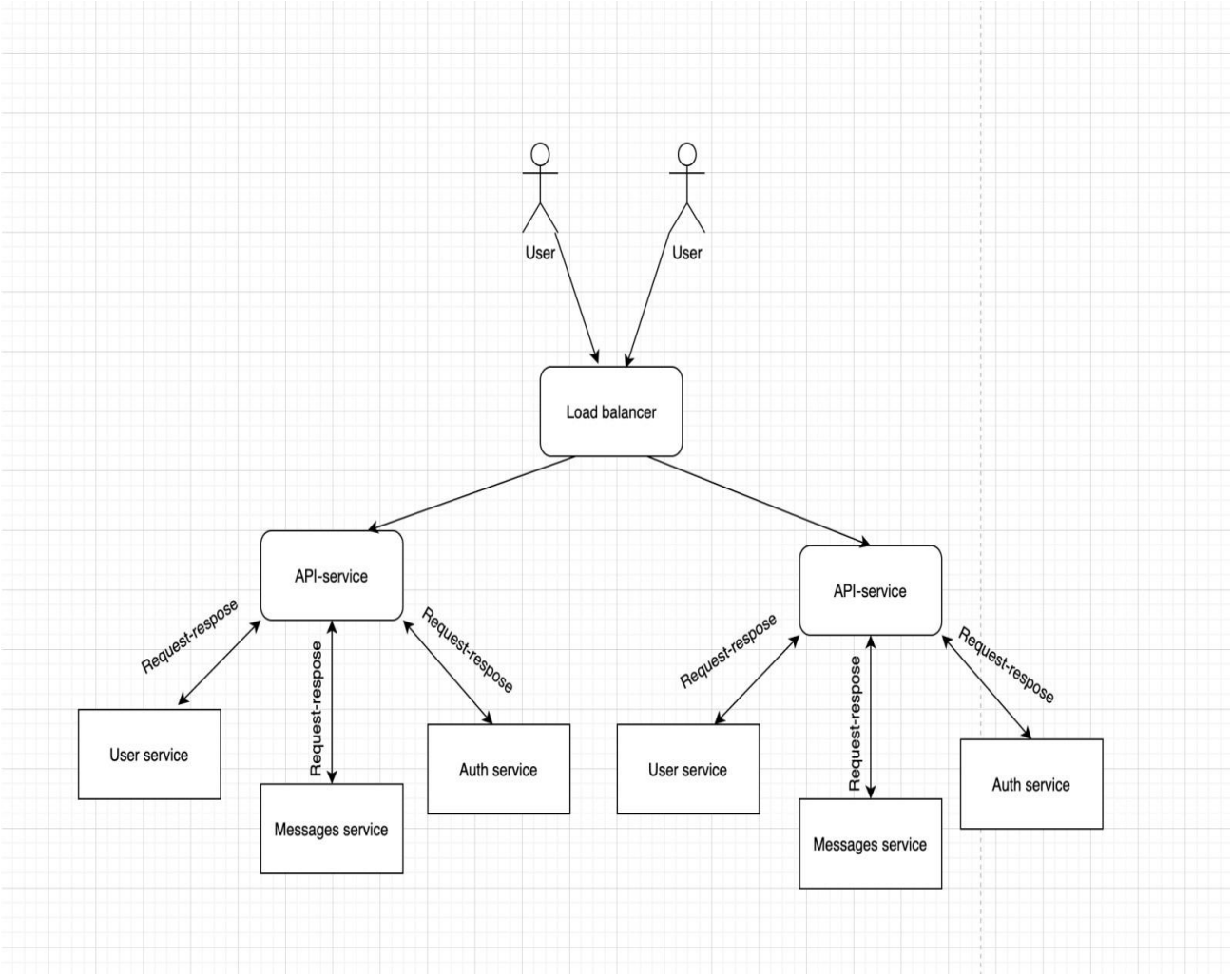


Рисунок 1.4 Масштабування мікросервісної архітектури

Другий спосіб – це розмножувати конкретний сервіс та встановлювати load-balancer перед ними. Load-balancer як і в першому випадку буде працювати за алгоритмом round-robin[7]. Структура даної системи показана на рис.1.5. Перевагою такого підходу є більша гнучкість в масштабуванні, адже розмножується мікросервіс на який йде найбільше навантаження, а ненавантажені мікросервіси так і залишаються в єдиному екземплярі. Отже такий підхід дозволяє суттєво економити ресурси. Єдиним слабким місцем стає вхід в систему – API-service, адже він залишається одним, тому якщо пропускна здатність API-service «гальмує» систему, потрібно розмножити його та встановити load-balancer.

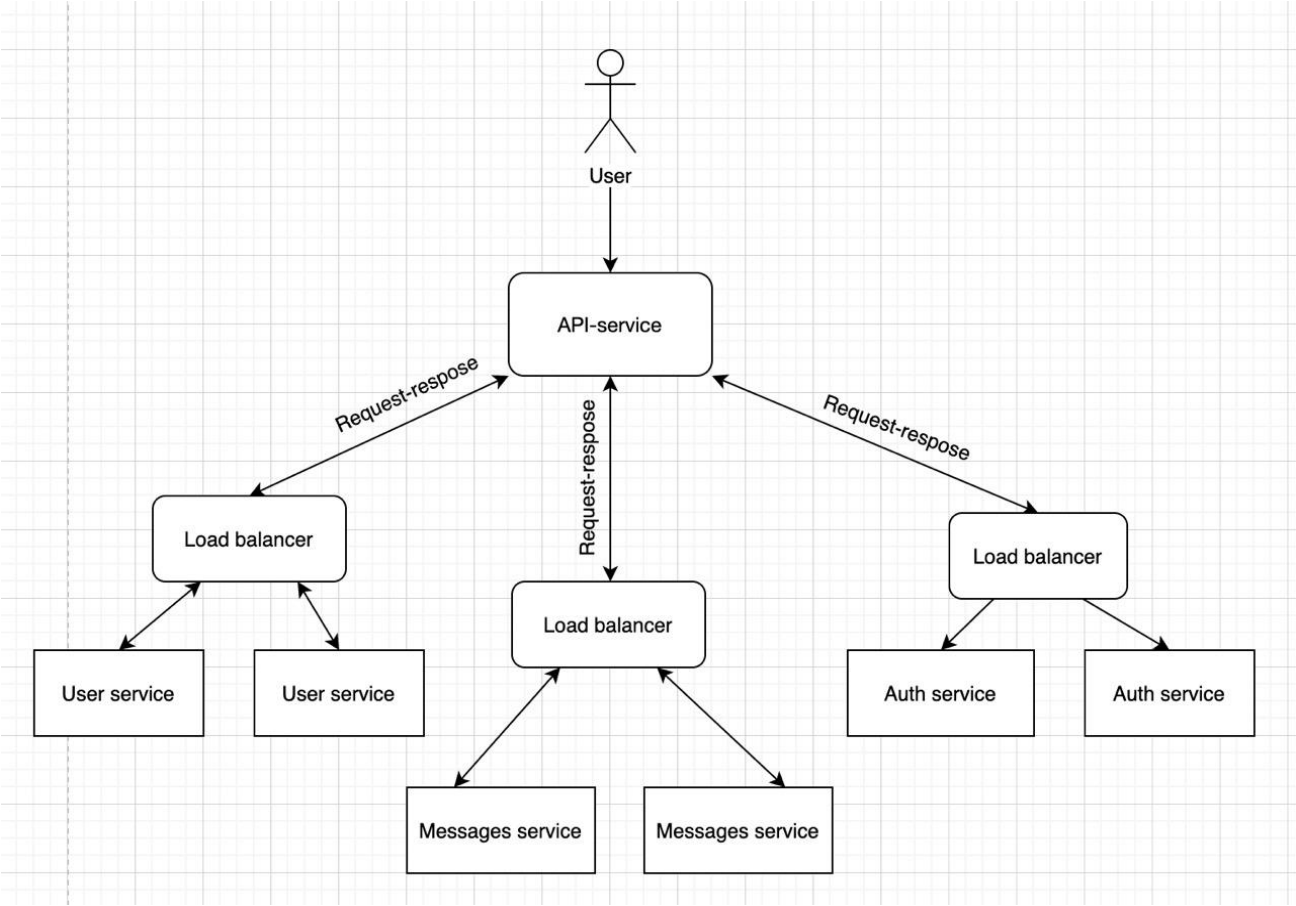


Рисунок 1.5 Масштабування мікросервісної архітектури

1.5 Використання cloud систем в розподіленій архітектурі сервера

На сьогоднішній день існує велика кількість сервісів, які надають можливість розгортати складні обчислювальні системи в хмарі. Іншими словами, такі сервіси здають в оренду обчислювальні потужності та програмне забезпечення, яке дозволяє легко його конфігурувати під необхідні цілі. Головними перевагами cloud систем є:

1. Глобальний масштаб. Хмарні сервіси дозволяють уникнути капітальних витрат на придбання обладнання та програмного забезпечення, налаштування і експлуатацію локальних центрів обробки даних, а це стійки з серверами, цілодобова подача електрики для живлення, охолодження та кваліфіковані IT-фахівці для управління цією інфраструктурою. Ці витрати швидко зростають[8].

2. Можливість масштабування. В контексті хмарних сервісів це означає виділення необхідного обсягу IT-ресурсів (наприклад, збільшення

або зменшення обчислювальної потужності, обсягу сховища або пропускної здатності) тоді, коли це потрібно, і в відповідному географічному розташуванні[8].

3. Потужність. Найбільші хмарні обчислювальні служби працюють в світовій мережі центрів обробки даних, які регулярно оновлюються до самого останнього покоління швидкого і ефективного обчислювального обладнання. Це забезпечує різні переваги в порівнянні з використанням одного корпоративного центру обробки даних, включаючи зменшення затримки в мережі і велику економію[8].

4. Безпека. Багато постачальників хмарних служб пропонують широкий набір політик, технологій і засобів контролю, які в цілому підвищують рівень безпеки, допомагаючи захистити дані, додатки і інфраструктуру від потенційних загроз[8].

5. Надійність. Хмарні сервіси зазвичай роблять резервне копіювання даних, мають механізми аварійного відновлення і роблять безперервність бізнес-процесів більш легкими і менш витратними.

6. Швидкість. Більшість хмарних сервісів працюють в режимі самообслуговування і за запитом, так, що навіть великі обсяги обчислювальних ресурсів можна запустити за кілька секнунд. Це дає клієнтам гнучкість і дозволяє позбутися постійного планування завантаження.

Розглянемо типи хмарних сервісів.

1. Модель IaaS включає в себе основні компоненти хмарних технологій. Вона надає доступ до мережевих можливостей, комп'ютерів (віртуальне або виділене обладнання) і певного обсягу сховища. IaaS передбачає максимальний рівень гнучкості і найширші можливості контролю ресурсів. Ця модель більш інших схожа на існуючі технології, звичні для більшості розробників.

2.Модель PaaS не вимагає управління базовою інфраструктурою (найчастіше обладнанням і ОС) і дозволяє зосередитися на розгортанні

додатків і управління ними. Це підвищує продуктивність роботи, адже більше не доводиться турбуватися про придбання матеріально-технічних ресурсів, займатися плануванням потужності, обслуговуванням ПО, установкою оновлень безпеки і виконувати інші трудомісткі завдання, необхідні для роботи додатків.

3. Модель SaaS являє собою готовий продукт, який запускається і управляється постачальником сервісу. Найчастіше під SaaS-рішеннями розуміють додатки для кінцевих користувачів (такі як веб-сайти електронної пошти). При виборі цієї моделі не потрібно турбуватися про те, як відбувається обслуговування сервісу і управління базовою інфраструктурою. Вирішити потрібно лише питання, як саме буде використовуватися той чи інший ПЗ.

Висновок до розділу №1

Давно відомо, що побудова високонавантаженої, розподіленої системи доволі складна та нетривіальна задача. Хоч на сьогоднішній день високонавантажені системи оточують нас, досі не існує встановлених підходів та чітких методологій з побудови таких систем. Також зрозуміло, що для побудови таких систем, в команді розробки повинні бути спеціалісти різних напрямків, таких як «комп'ютерні мережі», «бази даних», «безпека комп'ютерних систем», «серверне програмне забезпечення», «клієнтське програмне забезпечення», «тестування високонавантажених систем».

Проте, головним аспектом, який вирішує успішність всього програмного продукту, є правильно побудована архітектура системи. Адже від архітектури залежить, наскільки система буде надійною, яку пропускну здатність вона матиме, наскільки складно буде її підтримувати та змінювати в майбутньому. Давно є відомий той факт, що неправильна архітектура системи дуже швидко може «вбити» весь проект, адже її наслідки не можуть проявитись на початкових етапах розробки, а як правило, дають про себе знати з певним часом, коли вже внести зміни в архітектуру майже неможливо.

Отже, з першого розділу, можна зробити висновок, що архітектура системи є найважливішим аспектом в побудові високонавантаженої системи.

РОЗДІЛ 2
АНАЛІЗ АРХІТЕКТУРНИХ РІШЕНЬ ДЛЯ ПОБУДОВИ РОЗПОДІЛЕНИХ ВЕБ-ЧАТІВ

2.1 Основні вимоги до архітектури розподіленого веб-чату

Головною задачею даної системи є велика пропускна здатність, аби чат міг витримувати велике навантаження.

Отож опишемо головні вимоги до архітектури:

1. Розподілена архітектура, тобто система має складатись з багатьох серверів.

2. Можливість горизонтального масштабування системи, тобто аби ми могли легко збільшувати або зменшувати кількість серверів.

3. Надійність. Система повинна бути стійкою, втрата даних повинна бути мінімальною

4. Безпека. Система повинна мати механізми захисту від розповсюджених атак та не мати слабких місць з точки зору безпеки.

5. Швидкість. Система повинна мати мінімальний час затримки оброблення клієнтських запитів.

6. Механізми аварійної зупинки. В разі непередбачених подій в наслідок яких система зупиняється, всі данні що знаходяться в оперативній пам’яті повинні бути збережені, а процес операційної системи завершився максимально коректно. Після такої зупинки команда підтримки повинна бути сповіщена автоматично, а система намагалась відновитись.

Головною проблемою розподіленої архітектури в контексті веб-чату є те, що клієнти можуть під’єднатись до різних серверів і в такомму разі повідомлення потрібно пересилати між серверами аби вони досягли потрібного клієнта.

2.2 Опис основних архітектурних підходів

2.2.1 Шина повідомлень

Першим варіантом розподіленої архітектури є створення певного посередника - шини, яка буде займатись пересилкою даних (повідомлень, статусів…) з однго сервера до іншого. Дана система найлегша в реалізації, але має суттєвий недолік, оскільки шана в системі одна, пропускна здатність всієї системи буде обмежуватись шиною. Такий підхід нам не підходить, адже навантаження чату дуже швидко стане граничним для такої системи. Структура такої системи зображена на рис. 2.1

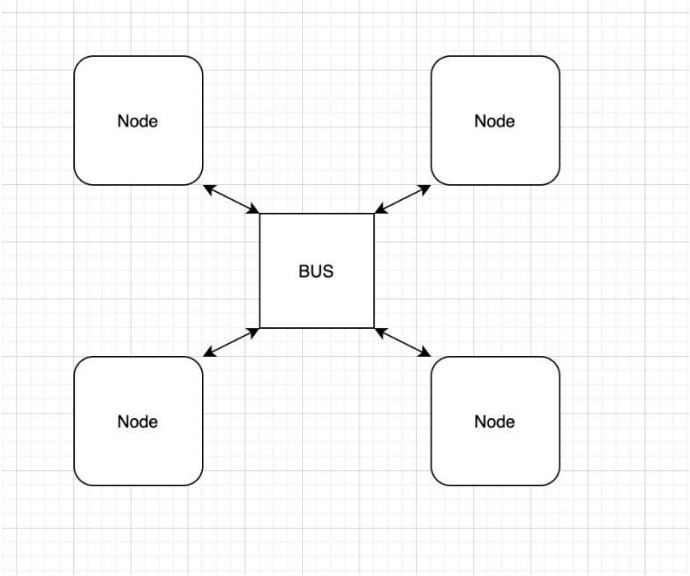


Рисунок 2.1 Розподілена архітектура з шиною повідомлень

Реалізацією такої архітектури може слугувати з’єднання всіх серверів(нод) між собою та до Master-сервера (Шини). Такий підхід дозволяє пересилати повідомлення напряму до потрібного сервера. Master-сервіс буде виконувати роль «арбітра», який буде знати про всі ноди та ділитись цією інформацією з кожним учасником кластеру. Під’єднання нової ноди в систему можна описати наступними кроками:

1. Під’єднання до Master-сервіса та інофрмування його про адресу нової ноди.

2. Master-сервіс оповіщає всі ноди системи про нового учасника

3. Новий сервер підключається до інших учасників кластеру

При відправці повідомлення сервер повинен розіслати це повідомлення по всім учасникам кластера, адже серверу до якого підключився відправник повідомлення не знає до якого сервера підключений отримувач.

Такий підхід має суттєвий недолік – перевантаження кожного сервера обробкою непотрібних даних, в виді пересилання та прийому надлишкових повідомлень. При отриманні нового повідомлення нода повинна проітеруватись по списку підключених клієнтів та перевірити чи серед них немає отримувача повідомлення, така операція доволі складна тому вона суттєво сповільнює ноду. Такий підхід не підходить для чату, адже розрахований на невелику кількість клієнтів. Структура такого підходу показана на рис. 2.2.

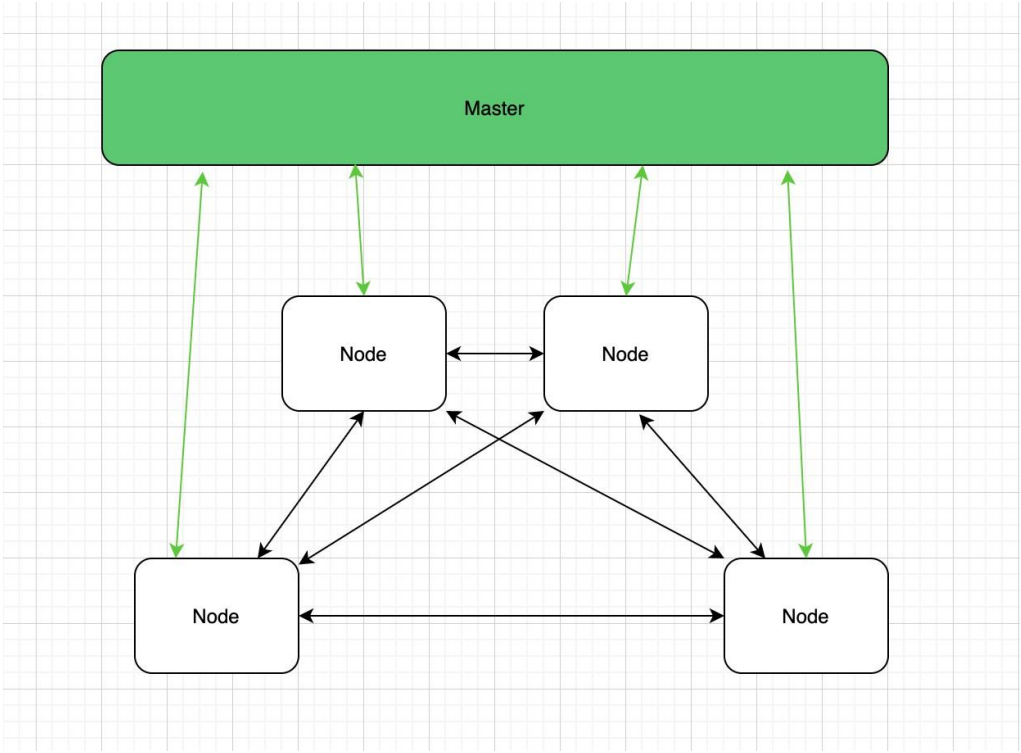


Рисунок 2.2 Система, яка розсилає повідомлення в кожну з нод

2.2.2 Сервер-арбітр

Одним з варіантів архітектури є з’єднання нод між собою та до Master-сервіса, при цьому Master-сервіс повинен знати, які корстувачі до якої ноди підкючені. При пересилці повідомлення кожна нода повинна

опитати Master-сервіс до якої ноди підключений отримувач повідомлення рис 2.3.

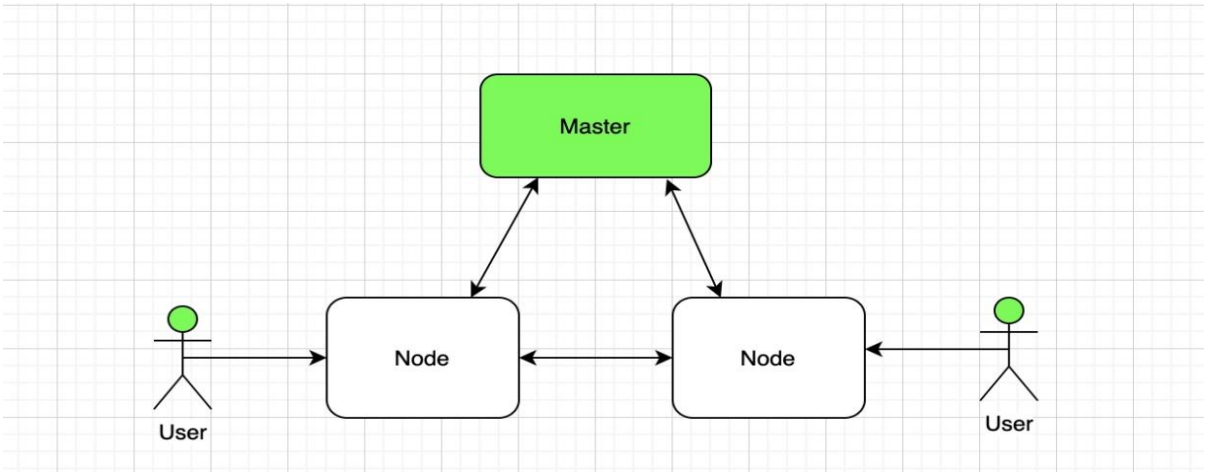


Рисунок 2.3. Розподілена система з Master-сервіс

Тоді процес пересилки повідомлення можна описати наступними кроками рис. 2.5:

1. Нода №1 опитує Master-сервіс до якої ноди під’єднаний отримувач повідомлення

2. Master-сервіс повідомляє ноді №1 про номер ноди (нариклад №3) до якої підключений отримувач

3. Нода №1 відслає повідомлення ноді №3.

4. Нода №3 відсилає повідомлення отримувачу.

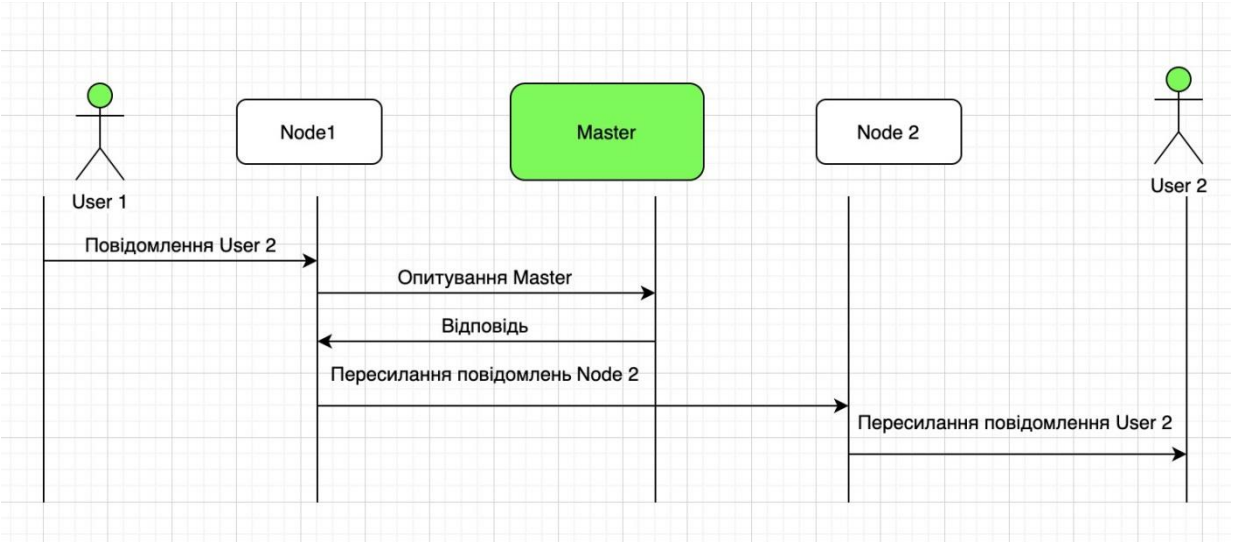


Рисунок 2.5. Кроки відправки повідомленнь

Перевагою такого підходу є пересилка повідомлення відразу потрібній ноді, що дозволяє уникнути надлишкової обробки даних в системі. Недоліком такого підходу є перевантаження Master-сервіс, адже при пересилці повідомлення кожна нода буде звертатись до Master-сервіса. Даний підхід є доволі хорошим для веб-чату, але все одно не має суттєві недоліки, які не дозволять йому горизонтально масштабуватись в майбутньому.

2.2.3 Пересилання напряму

Найкращим варіантом архітектури є підхід в якому кожна нода буде пересилати повідомлення напряму потрібній ноді без дублювання іншим учасника кластеру та без допомоги сервісів-посередників рис.2.6. Такий підхід можна реалізвати, якщо кожна нода буде сповіщати інших учасників кластеру про підключених клієнтів. Також при підключенні нової ноди, всі учасники кластеру повинні переслати списки підключених користувачів новій ноді. При зупинці роботи, нода повинна сповістити всіх підключених до неї клієнтів до якої ноди їм перепідключитись.

Даний варіант має певний недолік – це великий список підключених клієнтів, адже він буде містити інформацію про всіх учасників кластеру.

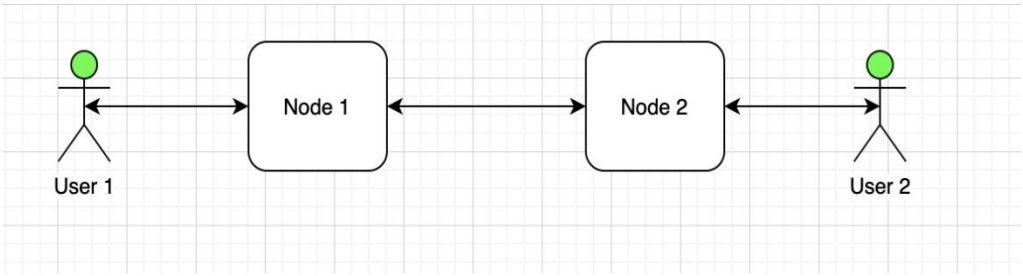


Рисунок 2.6 Пересилання напряму

Етапи пересилки повідомлення в системі можна описати наступною схемою рис. 2.7

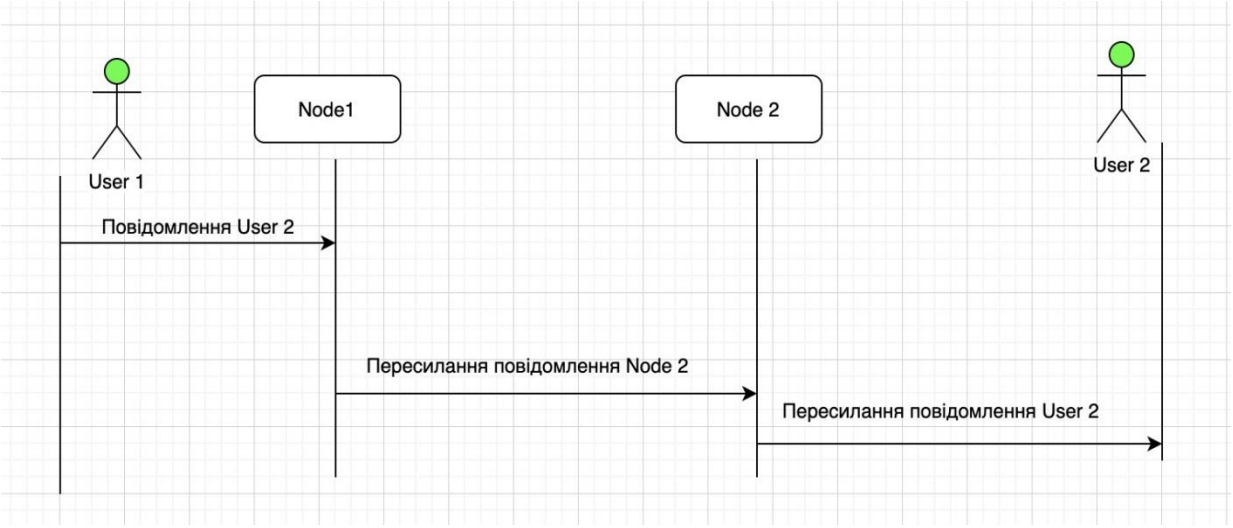


Рисунок 2.7 Етапи пересилки повідомлень

Отож, проаналізувавши всі архітектурні підходи було вибрано четвертий варіант адже він є найбільш перспективним з точки зору швидкодії системи.

2.3 Механізми пересилки даних.

На сьогодні є декілька варіантів програмного забезпечення що дозволяє пересилати данні між серверами:

2.3.1 RabbitMQ

RabbitMQ - це брокер повідомлень. основна мета RabbitMQ – отрмування та віддача повідомлень[9]. RabbitMQ система на базовому рівні містить наступні скалдові:

1. Producer (постачальник) – компонент, що займається відправкою повідомлень, на схемі позначено літерою «P»:

2. Queue (черга) – головний копонет RabbitMQ, всі повідомлення що передаються через RabbitMQ проходять через дану чергу. Черга ніяк не обмежена на кількість повідемлень, чергу можна сприймати як незкінченний буфер. Черга може приймати повідомлення від будь-якої кількості постачальників, також будь-яка кількість отримувачів може отримувати повідомлення від однієї черги. На схемі чергу позначено як «queue»

3.Consumer (отримувач) – компонент, що отримує повідомлення з черги.

Зазвичай отримувач знаходиться в стані очікування повідомлень. На схемі(рис.2.8) отримувач позначений літерою «С».

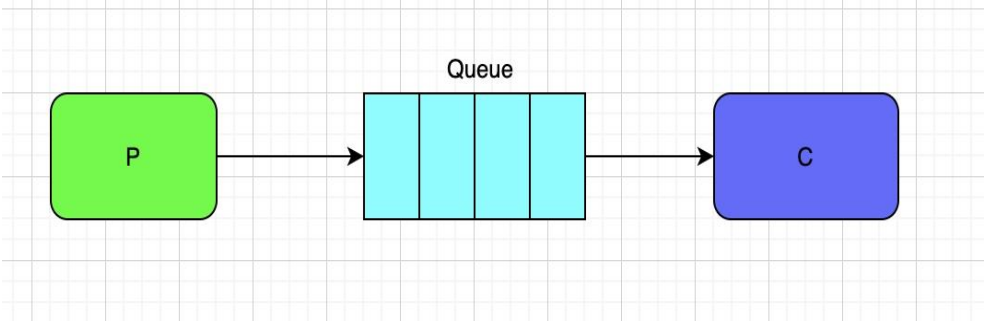


Рисунок 2.8 RabbitMQ

2.3.2 Kafka

Kafka – це брокер повідомлень, аналог RabbitMQ[10].

Kafka має три ключові можливості:

1. Публікація та підписка на потоки даних, подібні до черги повідомлень чи системи обміну повідомленнями в системі.

2. Зберігати данні які потрапляють в чергу в постійних сховищах, тобто повністю виключати можливість втрати даних

3. Обробляти відразу як вони потрапляють в чергу

2.3.3 Redis

Redis – система зберігання структур даних в оперативній пам’яті, яка використовується як база даних[11].

Redis також має вбудований механізм пересилки повідомлень, також відомий як «publisher-subscriber» (відправник-підписник)[12]. Даний механізм довзоляє створювати канали для пересилки даних, через які можна передавати дані від відправника до підписника. Реалізовано даний механізм доволі тривіально. В базі даних створюється спеціальна структура даних, наприклад масив, в який в майбутньому відправник буде вписувати дані, а підписник в свою чергу буде перевіряти присутність даних в цій структурі з певним інтервалом, наприклад 10 мілісекунд. Такий підхід не гарантує доставку даних, адже з певним інтервалом записи, які не були зчитані підписником видаляються. Суттєвою перевагою даного механізму є

надшвидка передача даних. Принцип роботи такого механізму показаний на рис 2.9

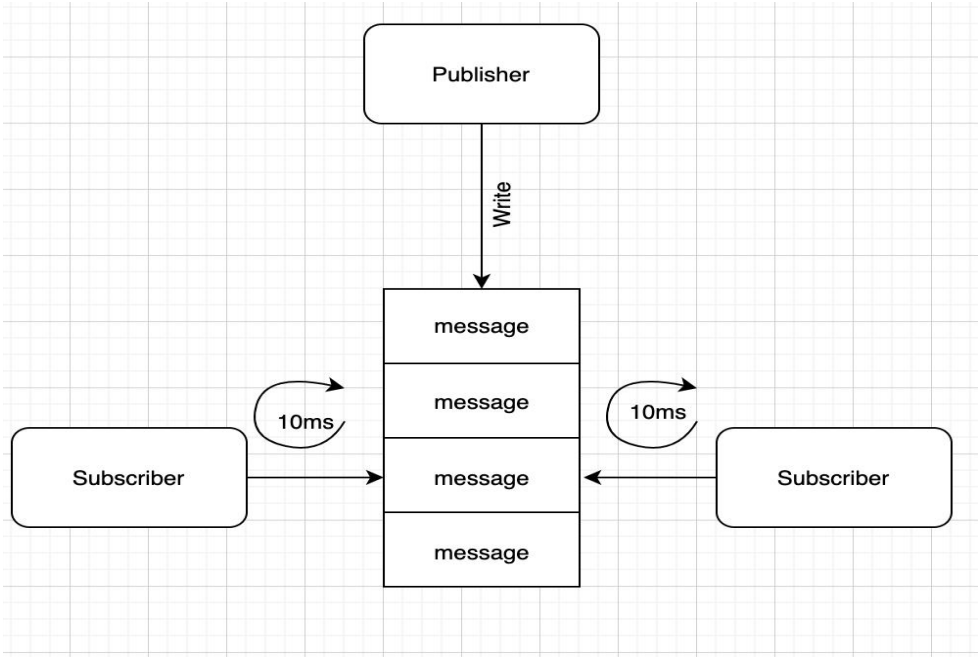


Рисунок 2.9 Принцип роботи механізму черг

Саме такий механізм підходить нам найбільше, адже він надає максимальну швидкодію, а проблемою негарантованої доставки можна знехтувати, адже даний підхід не буде використовуватись в системі в якій цілісність даних грає найважливішу роль, наприклад банківські чи фінансові системи, натомість, якщо одне з мільйона повідомлень в чаті втратиться нічого страшного не станеться. Для мінімізації втрати повідомлень всі данні будуть постійно копіюватись в постійні сховища. Отож при такому підході ми отримуємо наступні переваги:

1. Швидкодія за рахунок простого і надійного механізму Redis pub/sub
2. Гарантія доставки даних за рахунок використання мехінзму копіювання даних (реплікація)

2.4 варіанти використання Redis в росподіленій архітектурі веб-чату

Одним з підходів є зв’язання Redis з кожною нодою кластеру.

В разі відправлення повідомлення система відтворить наступні кроки(рис. 2.10):

1. Відправник посилає повідомлення в ноду до якої він підключений
2. Нода відправника пересилає повідомлення в Redis
3. Redis розсилає повідомлення по всім учасникам кластеру

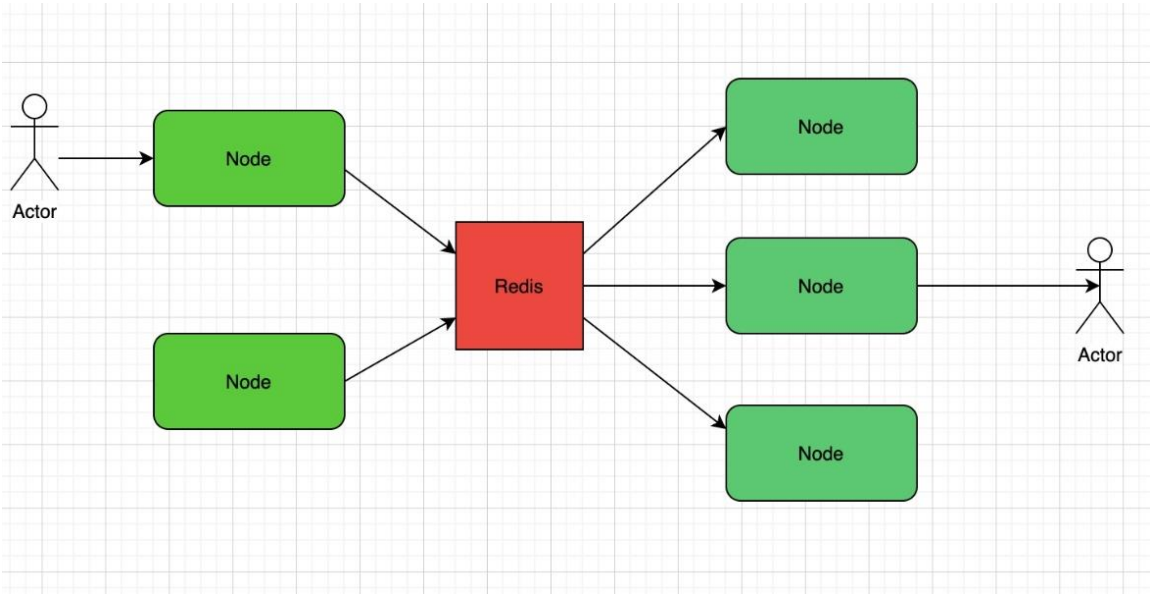


Рисунок 2.10 Кроки відправки повідомлення

Такий підхід обмежений в горизонтальному масштабуванні адже Redis тільки один і дуже швидко навантаження на нього зросте і він стане слабким місцем системи. Також до недоліків можна віднести той факт, що всі ноди будуть отримувати всі повідомлення, що не є ефективним використанням ресурсів. З вище перелічених суджень, можна зробити висновок що, такий підхід не підходить для високонавантаженого чату.

Для покращення надійності передачі даних, Redis об’єднують в певний кластер. При пересилці повідомлення, воно буде дублюватись в кожному Redis тож вірогідність втрати повідомлення суттєво знижується. Структура такої системи показана на рис. 2.11

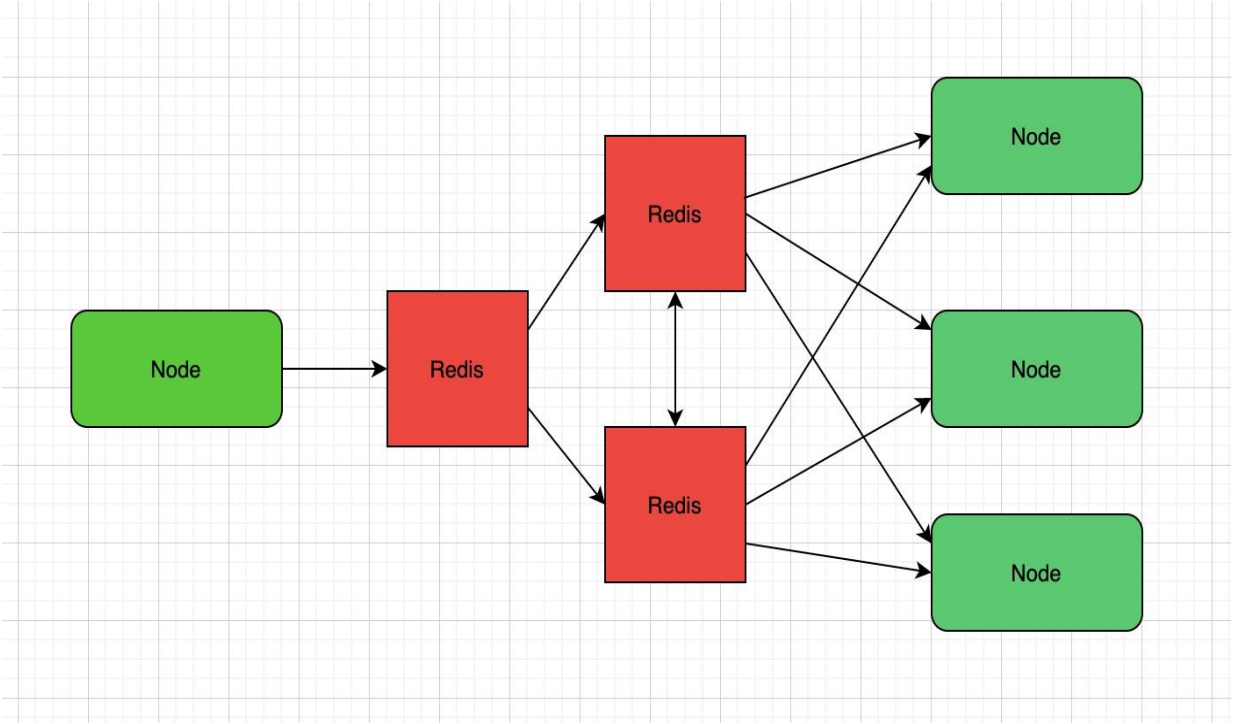


Рисунок 2.11 Системи з кластером Redis

Такий підхід хоч і дає приріст надійності, але має суттєвий недолік – подорожчання інфраструктури системи.

Підвищити пропускну здатність такої системи можна за допомогою наступного підходу: ввести в систему декілька Redis та дозволити ноді надислати повідомлення в один випадковий Redis і в свою чергу отримуваи повідомлення від всіх Redis(рис 2.12). З таким підходом система може розширювати Redis кластер[13], проте зростає навантаження на ноди.

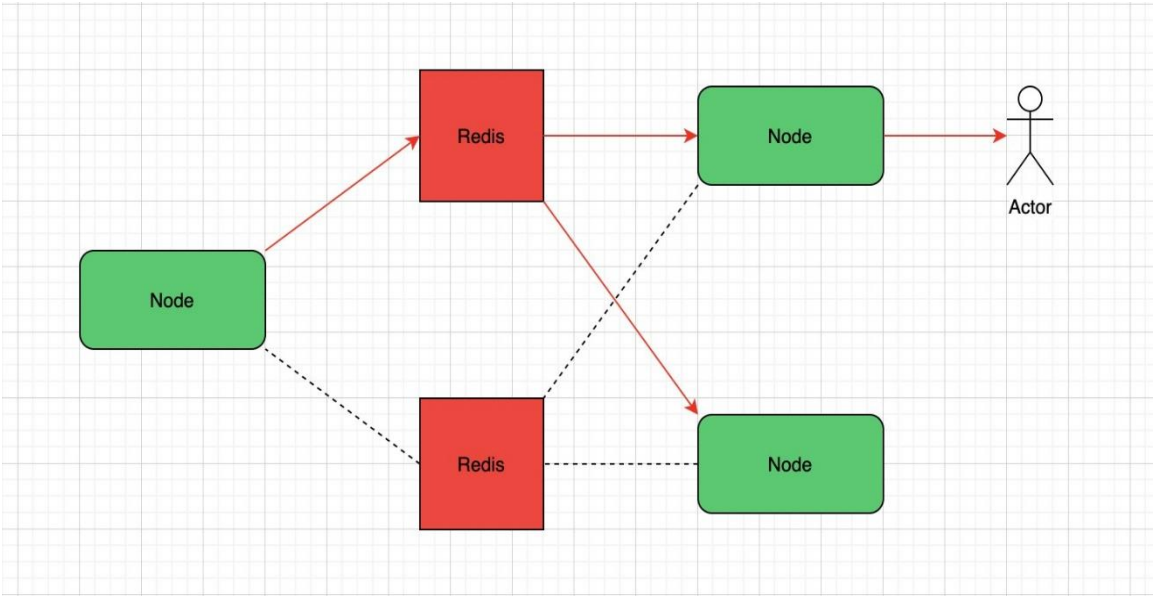


Рисунок 2.12 Покращена система з кластером Redis

Також можливий наступний варіант – нода підписується на випадковий Redis і при цьому пересилає повідомлення у всі Redis (рис 2.13). Такий підхід суттєво знімає навантаження з нод, але при цьому навантажує Redis кластер, адже він повинен пересилати повідомлення у всі ноди.

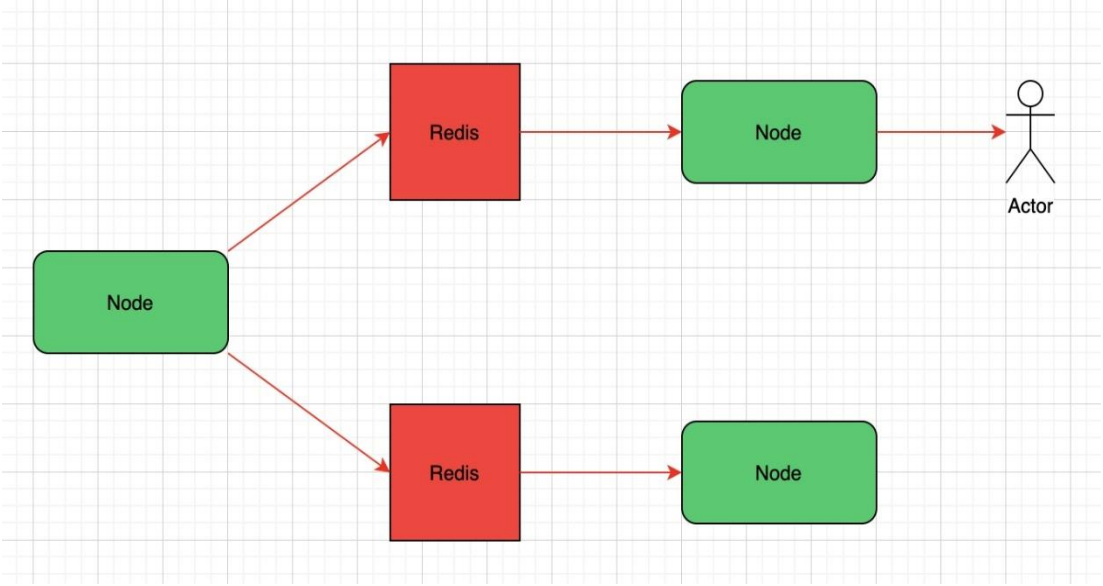


Рисунок 2.13 Покращена система з кластером Redis

Організувавши підписки наступним чином: кожному користувачу у відповідність буде створюватись канал на який нода зможе підписатись та отримувати тільки потрібні повідомлення. Таким чином буде зменшено навантаження на кожну з нод. Варто зауважити, що при такому підході

збільшується кількість підписок, що призводить до більшого використання пам’яті адже підписка або відписка від каналу доволі складна та ресурсоємна операція.

Для того аби надати такій системі можливість безграничного горизонтального масштабування було запропоновано наступний варіант (рис. 2.14): сервери Redis об’єднати в певні групи, наприклад по два сервера в кожну групу. Ноди будуть підписуватись на один Redis з кожної групи, а при пересилці повідомлення, будуть відсилати його у всі Redis з виподково вибраної групи.

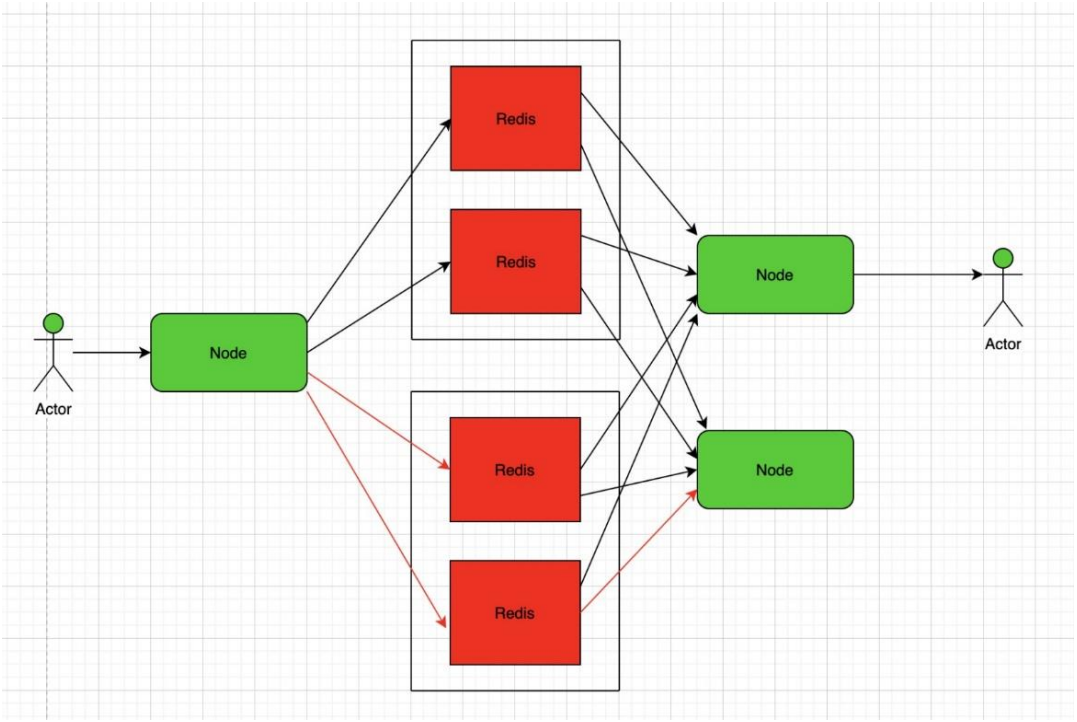


Рисунок 2.14 Redis об’єднані в групи

Відправку повідомлення в даній системі можна описати наступною схемою (рис. 2.15)

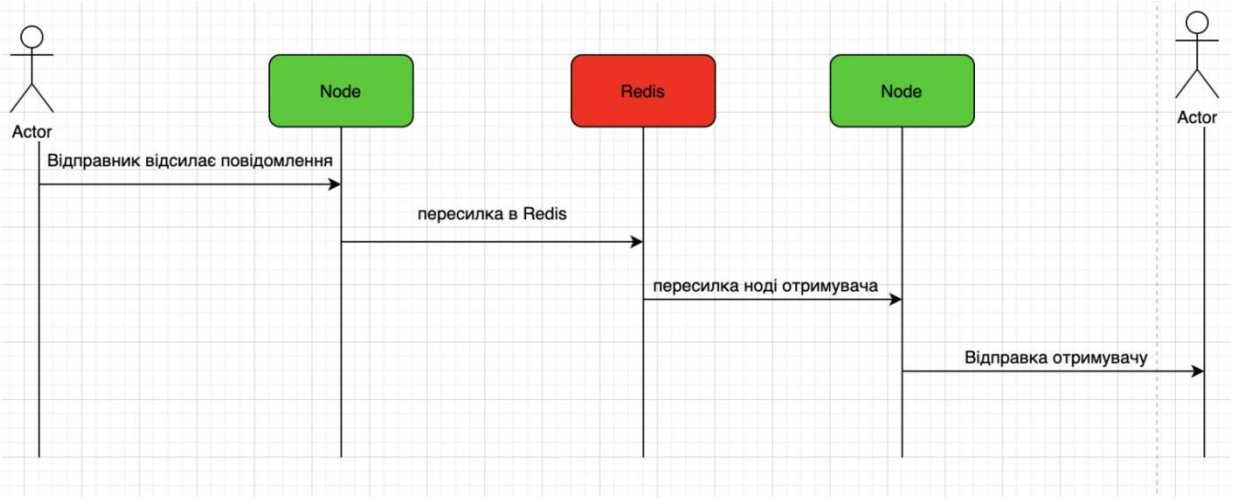


Рисунок 2.15 Етапи відправки повідомлення

2.5 Використання хешування в розподіленій архітектурі веб-чату

Хешування (гешування, англ. hashing) — перетворення вхідних даних будь-якої довжини у вихідний бітовий масив фіксованої довжини[14]. Такі перетворення також називаються хеш-функціями, або функціями згортання, а їхні результати називають хешем, хеш-кодом, хеш-сумою, або дайджестом повідомлення (англ. message digest).

Хеш-функція використовується зокрема у структурах даних — хеш-таблицях, широко вживаних у програмному забезпеченні для швидкого пошуку даних. Хеш-функції використовуються для оптимізації таблиць та баз даних за рахунок того, що у однакових записів однакові значення хеш-функції. Тобто в хеш-функції є скніченний діапазон значень, цей діапазон значень можна розподілити між Redis серверами.

Отже, використовуючи хешування унікального індетифікатора користувача, можливо визначати до якого Redis сервера підключений користувач. Приклад псевдокоду:

```
function getHash(key:string):integer //  Хеш-функція, яка повертає число
        const RedisClients: RedisClient[] = []//  Масив серверів Redis
        const userId = "some uid"// Унікальний індетефікатор
        const client = RedisClients [hash(userId) % RedisClients.length]
return client;
```

Отже, при такому підході система отримує можливість необмеженого горизонтального масштабування.

2.6 Використання протоколу WebSocket для обіну даними між клієнтом та сервером

Протокол WebSocket слугує для організації обміну даними між клієнтом та сервером через так зване «постійне з'єднання». Дані передаються в виді пакетів не прирівнаючи з'єднання. Даний протокол знайшов своє застосування в сервісах, які вимагають безперервного обміну даними, наприклад онлайн ігри, веб-чати і подібні[15].

Протокол WebSocket позначається як ws:// , а його зашифрована (протоколом TLS) версія – wss://. Протокол ws:// не тільки небезпечний, адже він взагалі не шифрується, а і поступається надійністю. До того ж сучасні браузери можуть не відкривати сайти на яких використовується ws:// замість wss://.Етапи встановлення з'єднання показані на рис. 2.16

Наведемо простий приклад заголовків для запиту по WebSocket ("wss: //somehost.com/chat")[15]:

GET /chat

Host: somehost.com

Origin: https://somehost.com.info

Connection: Upgrade

Upgrade: websocket

Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==

Sec-WebSocket-Version: 13

Де

• Origin – сторінка з якої встановлюється з'єднання по WebSocket

• Connection: говорить про намір клієнта змінити протокол

• Upgrade: говорить на який протокол клієнт хоче переключитись

• Sec-WebSocket-Key - випадковий послідовність(ключ), згенерований браузером для безпечного з'єднання.

• Sec-WebSocket-Version – версія WebSocket протоколу

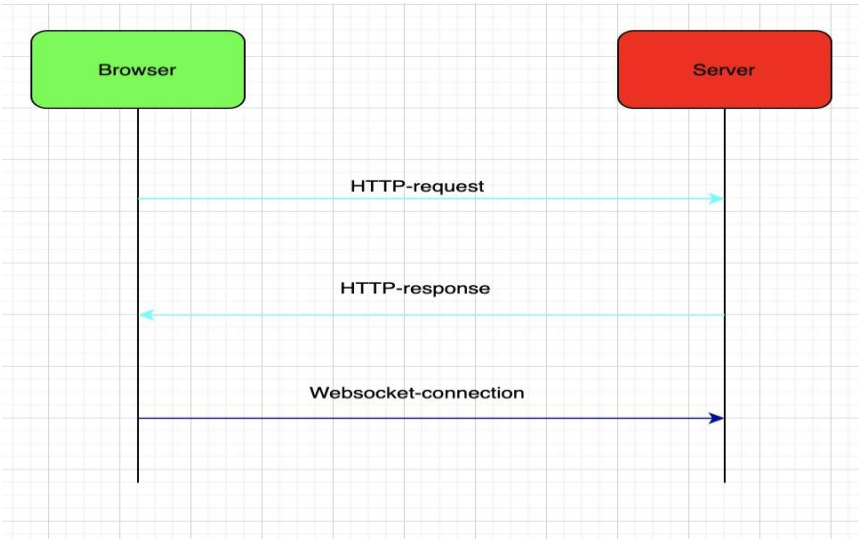


Рисунок 2.16 Етапи встановлення з’єднання по WebSocket

При згоді сервера перейти на WebSocket протокол, сервер має надіслати наступні заголовки:

101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=

Де Sec-WebSocket-Accept є Sec-WebSocket-Key, закодований спеціальним алгоритмом. Клієнт використовує дані заголовки аби запевнитись, що відповідь сервера співпадає з запитом. Після цього в дію вступає протокол WebSocket і дані передаються по ньому

Висновок до розділу №2

На сьогоднішній день існує багато підходів та технологій, які дозволяють як-завгодно організувати архітектуру, проте було розглянуто та проаналізовано основні способи організації серверної архітектури веб-чату. Після детального аналізу було вибрано найбільш вдалу архітектуру. Для організації кластеру було обрано підхід в якому для передачі даних між серверами використовується механізм Redis pub/sub, а для визначення до якого сервера підключився користувач використовується хешування. Такий підхід дозволяє передавати повідомлення на пряму між серверами, що суттєво заощаджує ресурси. Також перевагою такого підходу є необмежене горизонтальне масштабування.

РОЗДІЛ 3

РОЗРОБКА СИСТЕМИ

3.1 Розробка серверної частини

3.1.1 Опис використаних мов програмування та бібліотек

Як мову програмування для серверної частини було обрано JavaScript та спеціальну платформу для запуску JavaScript на сервері – Node.js .

JavaScript (JS) — динамічна, об'єктно-орієнтована прототипна мова програмування. Реалізація стандарту ECMAScript[16]. Найчастіше використовується для створення сценаріїв веб-сторінок, що надає можливість на стороні клієнта (пристрої кінцевого користувача) взаємодіяти з користувачем, керувати браузером, асинхронно обмінюватися даними з сервером, змінювати структуру та зовнішній вигляд веб-сторінки.

JavaScript класифікують як прототипну (підмножина об'єктно-орієнтованої), скриптову мову програмування з динамічною типізацією[16]. Окрім прототипної, JavaScript також частково підтримує інші парадигми програмування (імперативну та частково функціональну) і деякі відповідні архітектурні властивості, зокрема: динамічна та слабка типізація, автоматичне керування пам'яттю, прототипне наслідування, функції як об'єкти першого класу.

Мова JavaScript використовується для:

1. написання сценаріїв веб-сторінок для надання їм інтерактивності;
2. створення односторінкових веб-застосунків (React, AngularJS, Vue.js);
3. програмування на стороні сервера (Node.js);
4. стаціонарних застосунків (Electron, NW.js);
5. мобільних застосунків (React Native, Cordova);
6. сценаріїв в прикладному ПЗ (наприклад, в програмах зі складу Adobe Creative Suite чи Apache JMeter);
7. всередині PDF-документів тощо.

Node.js — платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript[17]. Засновником платформи є Раян Дал (Ryan Dahl). Якщо раніше Javascript застосовувався для обробки даних в браузері користувача, то node.js надав можливість виконувати JavaScript-скрипти на сервері та відправляти користувачеві результат їх виконання. Платформа Node.js перетворила JavaScript на мову загального використання з великою спільнотою розробників.

Node.js має наступні властивості:

1. асинхронна одно-нитева модель виконання запитів;
2. неблокуючий ввід/вивід;
3. система модулів CommonJS;
4. рушій JavaScript Google V8;

Для керування модулями використовується пакетний менеджер npm (node package manager).

Для взаємодії з Redis використвоується бібіліотека, яка являє собою програмною обгорткою над консольним клієнтом. Для взаємодії з мехінізмом pub/sub використовується наступні програмні абстракції:

1. redis.createClient() - який сворює клієнт, який дозволяє підписуватись на канали по яким пересилаються повідомлення.

2. client.on(“channelName”, function handler()), даний метод приймає на вхід назву каналу по якому будуть передаватись повідомлення і функцію-обробник, яка буде виконувати певні дії у відповідь на надходження повідомлення.

3. client.push(“channelName”, data), де “channelName” є назвою каналу в який потрібно передати повідомлення, data - данні, які потрібно передати.

Приклад коду використання pub/sub:

```
const subscriber = redis.createClient();
const publisher = redis.createClient();
let messageCount = 0;
subscriber.on("subscribe", function(channel, count) {
  publisher.publish("a channel", "a message");
});
subscriber.on("message", function(channel, message) {
  messageCount += 1;
});
```

Для хешування індетефікатора клієнта використовується бібліотека xxHash. xxHash - доволі швидкий алгоритм Hash, що хешує на межі швидкодії оперативної пам'яті.Код є доволі компактним, а хеші є повністю однаковими на всіх платформах[18].

Як логгер(бібліотека для форматування вивду в консоль) використувується бібліотека cli-color. Дана бібліотека дозволяє форматувати вивід різним шрифтом, кольором, додавати в вивід точний час, записувати вивід в файли. Великою перевагою даної бібліотеки є зручний вивід помилок. Приклад використання бібліотеки:

```
console.log(clc.red("Text in red")); для виводу кольорового тексту
console.log(clc.red.bgWhite.underline("Underlined red text")); вивід з фоном
console.log(clc.red("red " + clc.blue("blue") + " red")); текст різних кольорів
process.stdout.write(clc.move(-2, -2));  передвигання курсору
process.stdout.write(clc.move.to(0, 0)); курсор в конкретну точку
process.stdout.write(clc.move.up(2));  передвигання курсору вгору
process.stdout.write(clc.move.down(2));  передвигання курсору донизу
```

process.stdout.write(clc.move.right(2)); передвигання курсору вправо

process.stdout.write(clc.move.left(2)); передвигання курсору вліво

clc.columns(data[, options]) – дозволяє сформувати своєрідні колонки в консолі для виводу табличних даних, наприклад:

```
process.stdout.write(
        clc.columns([
                [clc.red("Name"), clc. red ("Proffesion"), clc. red
        ("Sallary")], ["Piter", "Lawyer", 3400],
                ["Jon", "driwer", 2000], ["Jan", "doctor", 3000]
        ])
);
```

Для взаємодії по WebScoket протоколу використовується бібіліотека Socket.IO. Дана бібліотека дозволяє організувати двосторонню передачу данних в режимі реального часу. В основі бібліотеки лежить протокол WebSocket, який працює поверх tcp або udp протоколу[19]. Дана біліотека охоплює серверну та клієнтську частину, тобто в основі даної біліотеки лежить node.js сервер, та спіціальний клієнтиський модуль, для взаємодії з node.js сервером. Бібліотеку можливо використвоувувати окремо як на сервері так і на клінті. За відсутності підтримки протоколу WebSocket біліотека переходить на спеціальний “long-pulling” режим, який дозволяє повторювати поведінку WebSocket за допомогою http протоеолу. Дана бібіліотека доволі надійна аже дозволяє створювати з”єднання навіть за викоиистання:

1. Проксі-сервера
2. Антивіруса чи персонального фаєрволу

Клієнтська чатина має вбудований механізм перепіделючення в умавах поганого підключення. Бібліотка дозволяє моніторити стасут підключення сервера чи клієнта за рахунок вбудованого мехінізму перевірки статусу[19]. В основі даного механізму лежить періодичне опитуавння сервера чи

клієнта. Дана бібліотека реалізує підтримку кімнат, тобто створення спеціальних груп-клієнтів, які будуть отривуати повідомлення одночасно.

Для контейнеризації використовується Docker. Docker – це спеціальне програмне забезпечення для полегшеного розгортання і управління програмними застосунками в операційних системах, що підтримують процес контейнеризації.

Контейнер - це певний модуль програмного забезпечення, який вміщає в себе код з усіма його залежності (рис. 3.1), в наслідок чого програма може працювати швидко та надійно в будь-якому середовищі чи операційній системі[20]. Відбиток (image) контейнера Docker - це невеликий за розміром, автономний, програмний модуль, який включає все необхідне для старту програми: код, всі залежності коду, потрібні пакети, бібліотеки системні інструменти, системні налаштування.

Відбитки контейнерів стають контейнерами під час старту, а у випадку контейнерів Docker - відбитки стають контейнерами, коли вони запускаються на Docker Engine[20]. Не дивлячись на якій платформі розгорнуто контейнер, контейнерне програмне забезпечення завжди працюватиме однаково, як на ОС Linux, так і на Windows. Контейнери ізолюють програмне забезпечення від свого оточення та забезпечують стабільність роботи.

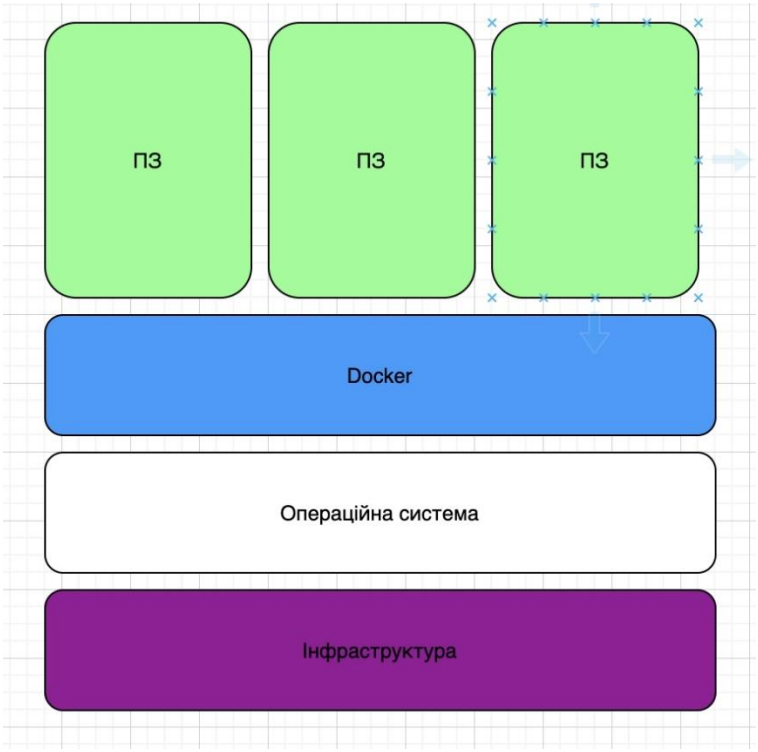


Рисунок 3.1 Контейнеризація застосунків

Для зручної роботи з консоллю використовується вбудований модуль Node.js - repl. Repl дозволяє описати команди та прикріпити до них функції-обробники. Приклад: repl.context.auth = (ruid) => {}; в даному прикладі опистується команда “auth”. Repl підтримує наступні команди за замовчуванням:

1. break: припинить подальше введення або обробку виразу, якщо він був введений

2. clear: повністю скидає контекст REPL і видаляє всі багаторядкові вирази, який знаходяться в процесі введення.

3. exit: викликаючи вихід REPL, закриває потік вводу-виводу,

4. help: відображає підказки пл спеціальним командам.

5. save: зберігає історію введення REPL у файл:> .save ./file/to/save.js

6. load: завантажує файл у поточний сеанс REPL. > .load ./file/to/load.js

7. editor: відкриває режим редактора (<ctrl> -D для закінчення, <ctrl> -C для скасування).

3.1.2 Взаємодія з Redis

Для описання взаємодії сервера з Redis було створено клас RedisController. Головним полем даного класу є масив “RedisClient”, який призначений для зберігання об”єктів підключення до Redis.

Опишемо ключові методи даного класу RedisController:

1. _getRedisClient призначений для ініціалізації об”єкту підключення до Redis, даний метод повертає екземпляр класу “RedisClient”

```
_getRedisClient(options: ClientOpts): RedisClient {
        const client = new RedisClient(options);
        client.on("error", (err) => console.error(err));
    return client;
}
```

2. getIndexOfServer даний метод приймає на вхід унікальний індетефікатор клієнта(користувача) і за допомогою хешування визначає індекс Redis сервера, до якого підключений клієнт. В ході хешування індетефікатора користувача отримується певне число, що потрапляє в один з діапазонів[18], які в свою черегу відповідають індексу певного Redis сервера.

```
getIndexOfServer(key: string | number): number {
        if (typeof key == "number") key = key.toString();
        return Math.floor(
            xxhash
            .hash64(Buffer.from(key, "utf8"), 0x2b0352df, "buffer")
            .readUInt32BE() % this.servers.length);
}
```

3. getServerByKey приймає на вхід індетифікатор користувача, виклакає метод getIndexOfServer і повертає підключення до Redis

4. Метод length повертає кількість підкючень до Redis

3.1.3 Обробка повідомлень

Для обробки повідомлень було реалізовано клас MessageController. В основі класу визначають три сновні поля:

1. subscribers - масив підключчень до Redis, що являють собою канали для отрмання повідомлень

2. Publishers - масив підключень до Redis, що є каналами для передачі повідомлень

3. clientsListenersList - структура данних, яка є масивом з хеш-мапів, яка призначена для зберігання функцій-обробників для кожного з каналів, по яким відбувається передача данних. Кожному Redis-підключенню в clientsListenersList відповідає хеш-мап, в якому в свою чергу для кожного каналу, що є ключем в відповідність ставиться масив функцій обробників.

Розглянемо процес ініціалізації хеш-мапів для кожного Redis підключення. Дану операцію виконує метод initListeners, який для кожного Redis підключення створює хеш-мап та підписує кожен канал Redis на подію "messageBuffer”, що викликається кожен раз коли по каналу приходить повідомлення.

```
initListeners(opts: ClientOpts[]) {
  for (let index = 0; index < opts.length; index++) {
    const eventListeners = {};
    this.clientsListenersList.push(eventListeners);
    this.subscribers
      .getServerByIndex(index)
      .on("messageBuffer", (buff: Buffer, data: Buffer) => {
        const passedChannel = buff.toString();
```

```
        if (!eventListeners[passedChannel]) return;
        eventListeners[passedChannel].forEach((c) =>
          c(msgpack.decode(data), passedChannel)
        );
      });
    }
  }
```

Спочатку дана функція в процесі ітерації створює стільки підключень, скіль вказано в конфігурації чату, а далі для кожного підключення формує хеш-мап і підписується на подію "messageBuffer". Обробник, що приймає повідомлення десереалізує повідомлення за допомогою msgpack, перш ніж віддати його на обробку.

Опишемо метод, який відповідає за приєднання до чату з іншим користувачем - метод join. Даний метод приймає на вхід індетифікатор каналу до якого потрібно приєднатись та функцію-обробник. За дпомогою хешування індетифікатору каналу, що відповідає унікальному індетефікатору користувача метод визначає до якого Redis сервера потрібно підключитися. Далі в масив clientsListenersList в потрібний хеш-мап додається функція-обробник, яка і буде обробляти повідомлення.

const clientIndex = this.subscribers.getIndexOfServer(channel);
визначення індексу Redis підключення.

this.subscribers.getServerByIndex(clientIndex).subscribe(channel);
підписка на отримання повідомлень по даному каналу.

const callbacksList = eventListeners[channel]; - масив функцій-обробників каналу

callbacksList.push(callback); - додавання функції-обробника

Опишемо метод покидання чату з іншим користувачем - метод leave. Метод приймає на вхід індетифіктор каналу та функцію обробник. Як і метод join, метод leave за допомогою хешування визначає потрібне Redis

підключення та хеш-мап з маисвами функцій-обробників, з яких і потрібно вилучити передану аргументом функцію-обробник.

const clientIndex = this.subscribers.getIndexOfServer(channel);

const eventListeners = this.clientsListenersList[clientIndex];

const callbacksList = eventListeners[channel]; - масив з функціями обробниками

const cb_index = callbacksList.indexOf(callback);

callbacksList.splice(cb_index, 1);

delete eventListeners[channel]; - вилучення функції-обробника.

Для відправлення повідомлення використовується метод publish. На вхді метод приймає індетифікатор каналу та данні, які потрібно передати. За допомогою хешування визначаєтьс яіндекс потрбіного Redis підключення. Використовується метод encode бібліотеки msgpack для сереалізації даних в бінарний формат.

const clientIndex = this.subscribers.getIndexOfServer(channel); - отрмання індексу Redis підключення.

this.publishers.getServerByIndex(clientIndex).publish(channel,

msgpack.encode(data)); - відправка повідомлення

3.1.4 Зберігання історії учасників чату

Історія учасників чатів зберігається в спеціальній структурі данних Redis, яка за власитивостями аналогічна хеш-мапу. Як ключ, якому відповідає історія повідомлень виступає унікальний інедетифкатор чату.

Для запису в даний хеш-мап використовується метод add, який приймає на вхід унікальний індетифікатор чату та індетифікатор користувача. Метод в свою чергу викликає метод Redis підключення - sadd. Для збереження історії учасників в потрібному Redis підключенні використовується метод getServerByKey, який за допомогою хешування визначає потрібне Redis підключення.

this.dialogsCache.getServerByKey(id).sadd(id, member);

Для видалення учасника з історії учасників чату використовується метод remove, який приймає на вхід унікальний індетифікатор чату та індетифікатор користувача. Метод в свою чергу викликає метод Redis підключення - srem. Для видалення повідомлення в потрібному Redis підключенні використовується метод getServerByKey, який за допомогою хешування визначає потрібне Redis підключення.

```
this.dialogsCache.getServerByKey(dialogId).srem(dialogId, member);
```

Для отрмання учасників чату використовується метод get, який приймає на вхід унікальний індетифікатор чату. Даний метод в свою чергу викликає метод Redis підключення - smembers, який дозволяє отрмати всі записи в хеш-мапі. Для того, аби отримати учасників з правильного Redisпідключення, використовується метод getServerByKey, який за допомогою хешування визначає потрібне Redis підключення.

```
return await new Promise<string[]>((resolve, reject) => {
        this.dialogsCache
        .getServerByKey(dialogId)
        .smembers(dialogId, (err, item) => (err ? reject(err) : resolve(item)));
});
```

Для того аби визначити чи є користувач в переліку користувачів певнгоо чату використовується метод getFor, який в приймає на вхід унікальний індетифікатор користувача та індетифікатор чату. Даний метод в сою чергу виклкає метод get та перевіряє його результат на присутність в ньому індетифікатора користувача.

```
const members = await this.get(dialogId);
if (!~members.indexOf(userId))
        throw new Error(`User "$forbidden to dialog "${dialogId}"`);
return members;
```

3.1.5 Взаємодія по WebSocket

Для взаємодії по WebSocket в межах кожної ноди створюється tcp сервер, який уде прийємати WebSocket підключення.

const io = SocketIO(3000, {}); - створення сервера

Процес підключення користувача до сервера описується в обробнику події "connection”

io.on("connection", (sock) => { … }).

В межах даного обробника стоврюється ціла низка обробників інших подій, таких як: “auth”, “logout”, “createChat”, “write”, “addMember”, “removeMember”, “disconnect”. Опишемо кожен з перелічених подій.

1. Подія “auth” викликається коли користувач реєструється в чаті під певним іменем. В межах даної функції створюється канал Redis, який в майбутньому буде приймати повідомлення від інших користувачів.

const uchan = toUserChan(uid);

const subs = await msgController.getNumOfSubs([uchan]);

if (!subs[uchan]) msgController.publish(toOnlineStatChan(uid), true);

msgController.join(uchan, receiveMessage);

2. Подія "logout" викликається коли користувач вирішив вийти з системи. В межах даної функції видаляються всі Redis канали по яким йшла передача даних та всі функції-обробники та оновлюється статус користувача.

msgController.leave(uchan, receiveMessage);

const subs = await msgController.getNumOfSubs([uchan]);

if (!subs[uchan]) msgController.publish(toOnlineStatChan(UID), false);

3. Подія “createChat” виликається, коли користувач створює чат. В межах даного метода створюється запис в історії учасників чату. В чат відправляється події про нового учасника.

dialogsController.add(data.dialog, UID); - стоврення запису в історії

msgController.publish(toUserChan(UID), { - відправлення події про нового учасника

```
    method: "memberAdded",
    data: {
      dialog: data.dialog,
      member: UID
    }
  });
```

4. Подія “write” виликається, коли користувач пише повідомлення чат. В межах даного чату отримується список користувачів за допомогою методу “getFor”. Для кожного користувача із списка, викликається метод publish

класу MessageController та передається повідомлення.

```
  const members = await dialogsController.getFor(UID, data.dialog);
  members.forEach((u) => {
    msgController.publish(toUserChan(u), {
      method: "write",
      data: data
    });
  });
```

5. Подія “addMember” виликається, коли користувач намагається долучитись до чату. В межах даного обробника отримується список користувачів за допомогою методу “getFor” та додається новий користувач за допомогою метода “add” класу dialogsController. Далі кожен користувач, який присутній в даному чаті отрмує сповізення.

```
const members = await dialogsController.getFor(UID, data.dialog);
```

отримання списку корисувачів

```
dialogsController.add(data.dialog, data.member);
```

- додавання нового користувача

```
[...members, data.member].forEach((u) => {
```

- сповіщення кожного окристувача

```
msgController.publish(toUserChan(u), {
```

```
            method: "memberAdded",
            data: data
            });
    });
```

5. Подія “removeMember” виликається, коли користувач намагається вийти зо чату. В межах даного обробника отримується список користувачів за допомогою методу “getFor” та видаляється користувач за допомогою метода “remove” класу dialogsController. Далі кожен користувач, який присутній в даному чаті отрмує сповіщення.

```
const members = await dialogsController.getFor(UID, data.dialog); - отримання списку корисувачів
dialogsController.remove(data.dialog, data.member); - видалення користувача
members.forEach((u) => { - сповіщення кожного окристувача
        msgController.publish(toUserChan(u), {
        method: "memberRemoved",
        data: data
        });
});
```

6. Подія "whoIsOnline" викликається, коли одни з коритсувачів намагається отримати список користувачів, які в даний момент часу знаходяться в чаті, тобто є онлайн. Список користувачів визначається за допомогою метода getNumOfSubs класу msgController та відсилається користувачеві, який зробив даний запит

```
const subs = await msgController.getNumOfSubs( - список користувачів в онлайні
        users.map((u) => toUserChan(u))
        );
resp(Object.keys(subs).map((uc) => fromUserChan(uc))); - відсилання відповіді користувачеві, який зробив даний запит
```

7. В межах події "disconnect" очищуються всі обробники іншиз подій

3.2 Розробка клієнтської частини

В основі клієнтської частини лежить робота з біліотекою Socket.io для обробки WebSocket та Repl, який дозволяє створювати консольні команди.

Websocket з'єднання створється за допомогою створення екземпляру класу Socket.io та передачі в конструктор url сервера.

```
const socketConnection = websocketClient(
 `http://localhost:${process.argv[2] || "3333"}`,
{transports: ["websocket"]});
```

Головною подією для WebSocket клієнта є подія connect, в момент коли вона стається на підключення "навішуються" й інші обробники подій, такі як: "write", "memberAdded", "memberRemoved", "statusChange". Розглянемо дані події:

1."write". Дана подія стається коли інший користувач в чаті відправляє повідомлення. Обробник даної події друкує повідомлення в консоль - console.log(`${clc.blue("Message")} from ${data.dialog} ==> ${data.msg}`).

2. "memberAdded". Дана подія стається коли інший користувач долучається до чату. Обробник даної події друкує в косоль, що інший користувач долучився в чат - console.log(`Member ${data.member} was ${clc.green("added")} to room ${data.dialog}`)

3. "memberRemoved". Дана подія стається коли інший користувач виходить з чату. Обробник даної події друкує в косоль, що інший користувач вийшов з чату - console.log(`Member ${data.member} was ${clc.green("removed")} from ${data.dialog}`)

4. "statusChange". Дана подія стається коли інший користувач змініює статус "онлайн". Обробник даної події друкує в косоль, що інший користувач змінив свій статус "оналайн" - console.log(`User ${data.user} is ${data.online ? clc.green("online") : clc.green("offline")} now`)

Перелічені обробнки події ініціалізуються в функції - “initSocketEventListeners”, яка в свою чергу викликається в обробнику події “connect”

```
socketConnection.on("connect", (err) => {
        console.log(clc.green("Client connected !!!"));
        initSocketEventListeners();
        initCommandLineInterface();
});
```

Функція initCommandLineInterface ініціалізує консольні команди, такі як: “auth”, “logout”, “subs”, “online”, “write”, “createChat”, “join”, “remove”. Кожна з перелічених команд має свою функцію-обробник, яка в свою чергу посилає відповідну подію по WebSocket. Розглянемо це на прикладі команди “write” та “createChat”:

```
repl.context.write = (dialog, msg) => {
  socketConnection.emit("write", { - відправка події "write"
   dialog: dialog.toString(), - унікальний індетифікатор діалогу(чату)
   msg: msg.toString() - повідомлення
  });
 };
repl.context.createChat = (dialog) => {
 socketConnection.emit("createChat", { - відправка події "createChat"
  dialog: dialog.toString() - унікальний індетифікатор діалогу(чату)
 });
};
```

Всі інші обробнки команд працюють аналогічним чином

Висновок до розділу №3

В даному розділі було описано написання клієнтської та серверної частини. Як мову програмування було обрано JavaScript, як на серверній, так і на клієнтській частині. Клієнт та сервер працюють на платформі Node.js.

В основі архітектури серверної частини лежить розподілений кластер, ноди якого спілкуються за допомогою протоколу Redis pub/sub. Для зберігання історії підключень використовується структура даних Redis – «хеш-мап». Було використано хешування унікального індетифікатора клієнта для визначення до якої ноди він під’єднався.

Для передавання даних між сервером та клієнтом було обрано протокол WebSocket та використано бібліотеку Socket.io, як на сервері, так і на клієнті.

Інтерфейсом клієнтської частини слугує набір консольних команд. Для створення консольних команд було використано вбудований node.js модуль «repl».

РОЗДІЛ 4
ТЕСТУВАННЯ СИСТЕМИ

Для запуску сервера потрібно ввести команду «docker-compose up» . Після чого серверна інфраструктура з двох нод та кластера Redis заупститься(рис 4.1).

```
user@Admins-MacBook-Pro distributed-nodejs-chat-with-redis % sudo docker-compose -f "dock
er-compose.yml" up -d --build
Creating network "distributed-nodejs-chat-with-redis_esnet" with the default driver
WARNING: Found orphan containers (distributed-nodejs-chat-with-redis_dialogs-cache_1, dis
tributed-nodejs-chat-with-redis_msg-bus-0_1, distributed-nodejs-chat-with-redis_msg-bus-1
_1) for this project. If you removed or renamed this service in your compose file, you ca
n run this command with the --remove-orphans flag to clean it up.
Creating distributed-nodejs-chat-with-redis_message_bus_1_1 ... done
Creating distributed-nodejs-chat-with-redis_message_bus_0_1 ... done
```

Рисунок 4.1 Запуск серверної частини

Для запуску клієнської частини потрібно ввести команду ts-node src/client.ts та аргуметом передати порт ноди до якої їоче під”єднатись користувач, наприклад 3001 (рис. 4.2).

```
user@Admins-MacBook-Pro sock-app % ts-node src/client
.ts 3001
Client connected !!!
>
```

Рисунок 4.2 Запуск клієнтської частини

Для авторизації користувача в системі потрібно ввести команду “auth” та передати аргуметом ім”я користувача. Зареєструємо два користувача - Simon та Fedir (рис. 4.3, рис 4.4)

```
user@Admins-MacBook-Pro sock-app % ts-node src/client
.ts 3001
Client connected !!!
> auth("Simon")
> Authorized as Simon
```

Рисунок 4.3 Авторизація користувача Simon

```
user@Admins-MacBook-Pro sock-app % ts-node src/clie
nt.ts 3002
Client connected !!!
> auth("Fedir")
> Authorized as Fedir
```

Рисунок 4.4 Авторизація користувача Simon

Створимо чат з назвою “First chat” та додамо в нього користувачів Simon та Fedir. Для створення чату нам потрібно ввести команду “createChat” та передати назву чату аргуметом (рис. 4.5).

```
> createChat("First chat")
> Member Simon was added to room First chat
```

Рисунок 4.5 Створення чату

Для того аби додати користувача “Fedir” до чату потрібно ввести команду “join” та передати назву чату та ім’я користувача аргуметами (рис. 4.6).

```
> join("First chat", "Fedir")
> Member Fedir was added to room First chat
```

Рисунок 4.6 Додавання користувача до чату

Користувач “Fedir” отримає сповіщення, про те, що його було додано до чату (рис. 4.7).

```
> auth("Fedir")
> Authorized as Fedir
Member Fedir was added to room First chat
```

Рисунок 4.7 Сповіщення про долучення до чату

Для того аби надіслати повідомлення користувачеві потрібно ввести команду “write” та передати назву чату та повідомлення аргументами (рис. 4.8).

```
> write("First chat", "Hi Fedir")
> Message from First chat ==> Hi Fedir
```

Рисунок 4.8 Написання повідомлення в чат

Корситувач Fedir отрмимає надіслане йому повідомлення (рис. 4.9).

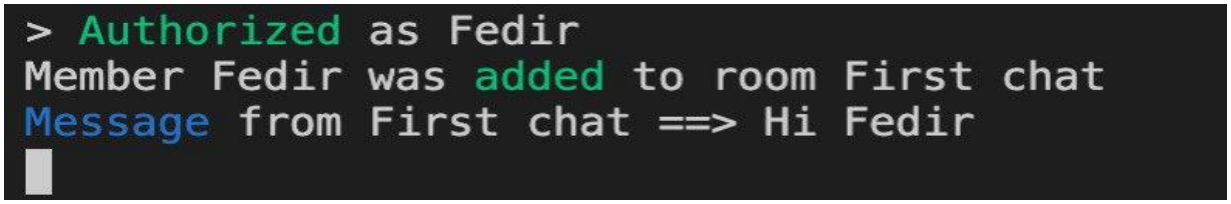


Рисунок 4.9 Отримання повідомлення

Для перевірки користувача на статус “онлайн” потрібно передати список користувачів на команду online. Перевіримо статус двох користувачів від імені Simon, одного автоизованого(“Fedir”), а іншого(“Petro”) - ні. Команда поверне користувача Fedir (рис. 4.10).

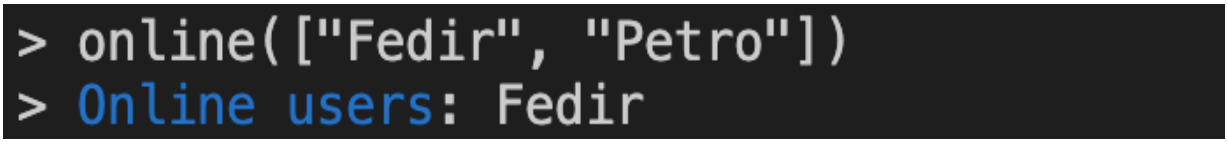


Рисунок 4.10 Перевірка статусу “онлайн”

Зробимо аналогічну перевірку від імені Fedir, перевіримо статус користувача Simon (рис. 4.11).

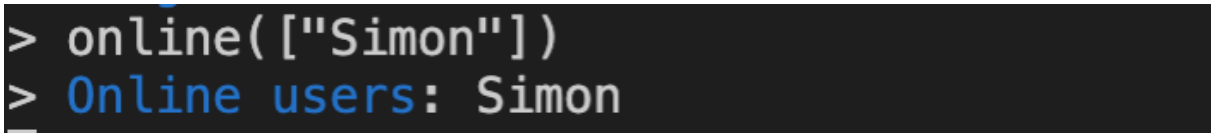


Рисунок 4.11 Перевірка статусу “онлайн”

Для підписки на зміну статуса “онлайн” використовується команда “subs”. Аргументами передається список користувачів. Підпишемось на зміну статуса “онлайн” користувача Fedir (рис. 4.12).

Рисунок 4.12 Підписка на зміну статуса “онлайн”

Від’єднаємо користувача Fedir від сервера (рис 4.13), в свою чергу користувач Simon отримає сповіщення про зміну статуса “онлайн” в користувача Fedir (рис. 4.14).

Рисунок 4.13 Від'єднання користувача

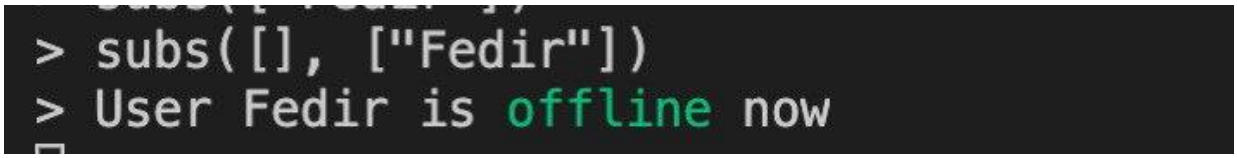


Рисунок 4.14 Отрмання сповіщення про зміну статуса “онлайн”

Висновок до розділук №4

У даному розділі було описано як користуватись інтерфейсом веб-чату. Роль інтерфейсу виконує клієнтський застосунок, який керується консольними командами. Команди виконують такі функції як: авторизація, створення чату, написання повідомлення в чат, визначення статусу «онлайн» користувача, підписка на зміну статусу «онлайн» користувача. Було перевірено пропускну здатність даної системи за допомогою синтетичних тестів, в основі яких лежить авторизація та написання повідомлення в новий чат. За результатами тесів, на локальному комп'ютері, дві ноди та Redis витримують до 1500 одночасних користувачів, що є доволі хорошим результатом, оскільки система на монолітному сервері витримувала 1100 підключень. Отже, було створено систему, яка виконує основні функції, які відведені для різного роду веб-чатів.

Даний програмний застосунок може слугувати основою для більш предметних систем, які можуть використовуватись в системах різного плану та масштабу. Архітектура даної системи дозволяє легко інтегрувати її у вже існуючі системи, не примушуючи вносити серйозі зміни в їх структуру.

ВИСНОВКИ

У роботі було розглянуто та доведено, що головним аспектом, який вирішує успішність всього програмного продукту, є правильно побудована архітектура системи.

По-друге, для організації кластеру було обрано та детально описано підхід, в якому для передачі даних між серверами використовується механізм Redis pub/sub, а для визначення до якого сервера підключився користувач використовується хешування. Критерієм вибору слугувала можливість передавати повідомлення на пряму між серверами, що суттєво заощаджує ресурси, та наявність необмеженого горизонтального масштабування.

По-третє, за допомогою мови програмування JavaScript та з використанням платформи Node.js було розроблено архітектуру серверної частини, в основні якої лежить розподілений кластер, ноди якого спілкуються за допомогою протоколу Redis pub/sub. Для передавання даних між сервером та клієнтом було обрано протокол WebSocket та використано бібліотеку Socket.io.

По-четверте, роль інтерфейсу виконує клієнтський застосунок, який керується консольними командами, що виконують такі функції як: авторизація, стоврення чату, написання повідомлення в чат, визначення статусу «онлайн» користувача, підписка на зміну статусу «онлайн» користувача.

Отже, архітектура даної системи дозволяє легко інтегрувати її у вже існуючі системи, не примушуючи вносити серйозі зміни в їх структуру.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ:

1. Representational state transfer. [Електронний ресурс]. – Режим доступу: URL: https://en.wikipedia.org/wiki/Representational_state_transfer (Дата звернення 21.05.2020).

2. Remote procedure call. [Електронний ресурс]. – Режим доступу: URL: https://en.wikipedia.org/wiki/Remote_procedure_call (Дата звернення 20.05.2020).

3. Pattern: Monolithic Architecture. [Електронний ресурс]. – Режим доступу: URL: https://microservices.io/patterns/monolithic.html (Дата звернення 12.05.2020).

4. Simon Wahlstrom Comparing scaling benefits of Monolitic and Microservices architectures implemented in Java and Go

5. Microservices. [Електронний ресурс]. – Режим доступу: URL: https://en.wikipedia.org/wiki/Microservices (Дата звернення 11.05.2020).

6. Pattern: Microservice Architecture. [Електронний ресурс]. – Режим доступу: URL: https://microservices.io/patterns/microservices.html (Дата звернення 11.05.2020).

7. Scaling Microservices: The Challenges and Solutions. [Електронний ресурс]. – Режим доступу: URL: https://dzone.com/articles/scaling-microservices-the-challenges-and-solutions (Дата звернення 12.05.2020).

8. Cloud Computing. [Електронний ресурс]. – Режим доступу: URL: https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/#benefits (Дата звернення 26.05.2020).

9. rabbitMQ. [Електронний ресурс]. – Режим доступу: URL: https://www.rabbitmq.com/documentation.html (Дата звернення 26.05.2020).

10. Introduction Kafka. [Електронний ресурс]. – Режим доступу: URL: https://kafka.apache.org/intro (Дата звернення 26.05.2020).

11. Redis. [Електронний ресурс]. – Режим доступу: URL: https://redis.io/ (Дата звернення 15.05.2020).

12. Redis pub/sub. [Електронний ресурс]. – Режим доступу: URL: https://redis.io/topics/pubsub (Дата звернення 23.05.2020).

13. Redis Cluster Specification. [Електронний ресурс]. – Режим доступу: URL: https://redis.io/topics/cluster-spec (Дата звернення 18.05.2020).

14. Хеш-функція. [Електронний ресурс]. – Режим доступу: URL: https://uk.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%8F (Дата звернення 16.05.2020).

15. The WebSocket Protocol. [Електронний ресурс]. – Режим доступу: URL: https://tools.ietf.org/html/rfc6455 (Дата звернення 16.05.2020).

16. JavaScript. [Електронний ресурс]. – Режим доступу: URL: https://uk.wikipedia.org/wiki/JavaScript (Дата звернення 15.05.2020).

17. Node.js. [Електронний ресурс]. – Режим доступу: URL: https://uk.wikipedia.org/wiki/Node.js (Дата звернення 15.05.2020).

18. xxHash. [Електронний ресурс]. – Режим доступу: URL: https://github.com/Cyan4973/xxHash (Дата звернення 25.05.2020).

19. Socket.io. [Електронний ресурс]. – Режим доступу: URL: https://socket.io/get-started/chat/ (Дата звернення 24.05.2020).

20. What is a container. [Електронний ресурс]. – Режим доступу: URL: https://www.docker.com/resources/what-container (Дата звернення 21.05.2020

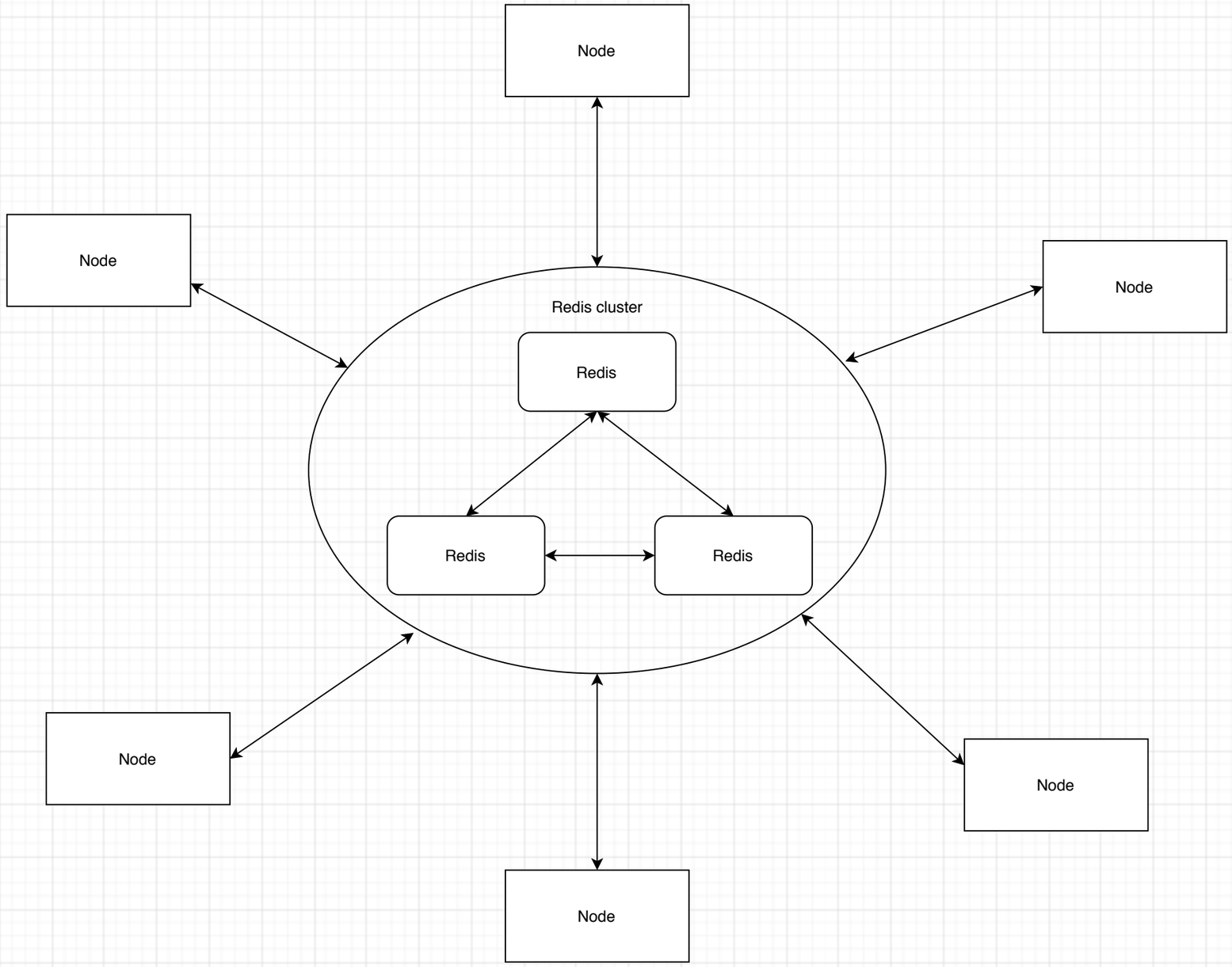
ДОДАТОК А

ЧАТ НА ОСНОВІ ВЕБ-СЕРВІСУ

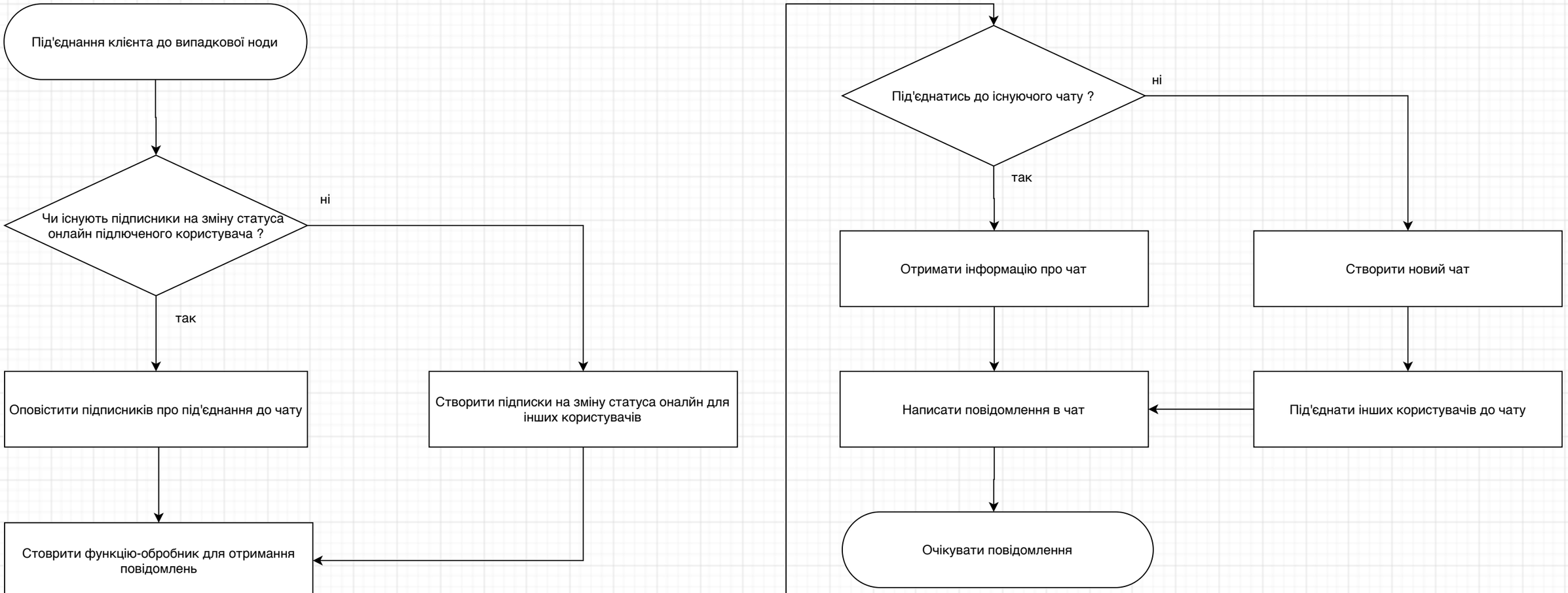
КОПІЇ ГРАФІЧНИХ МАТЕРІАЛІВ

Аркушів 3

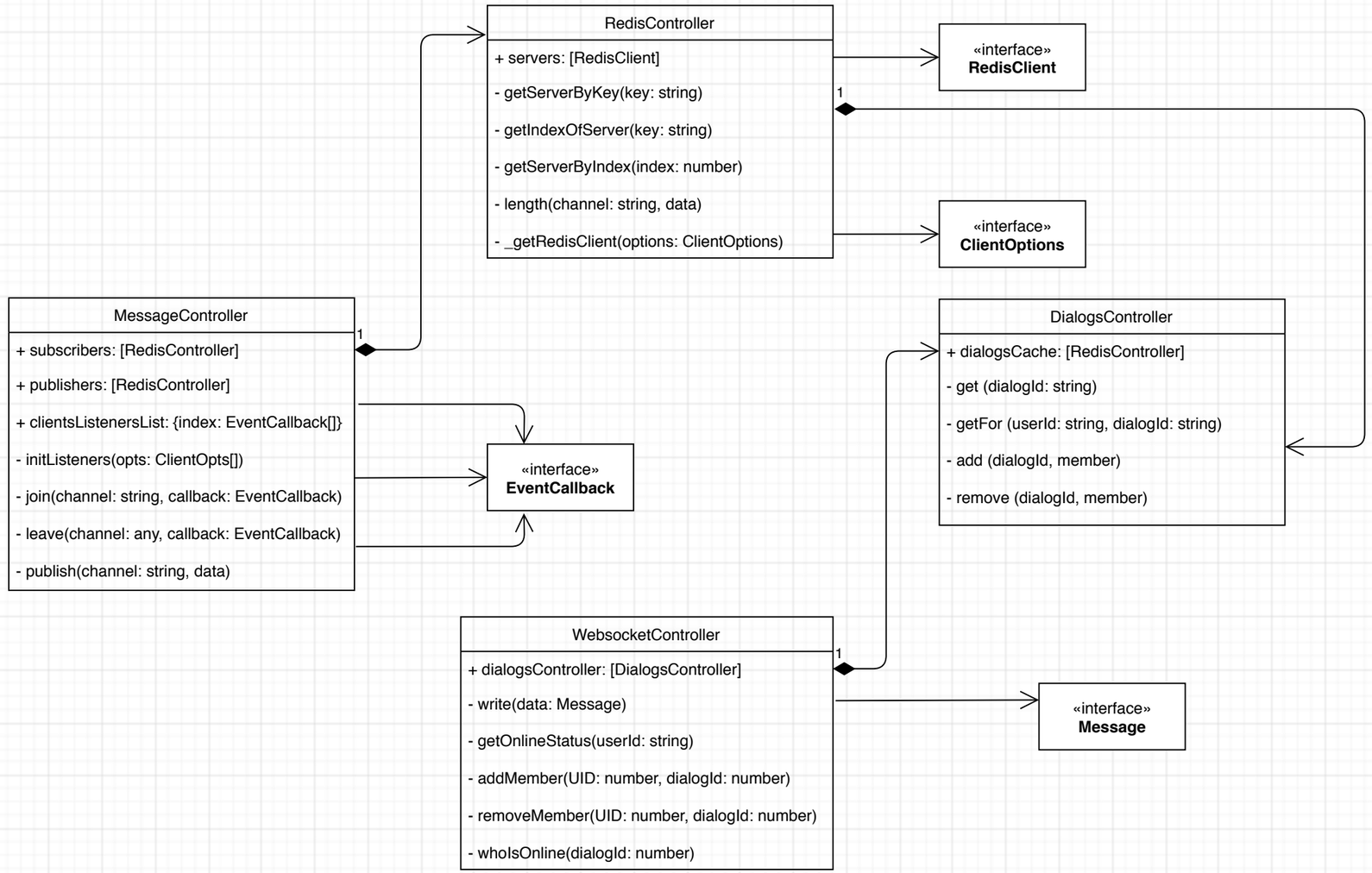
Київ – 2020



					ІАЛЦ.467100.04 Д1	
Змн.	Арк.	№ докум.	Підпис	Дата		
Розроб.		Коваленко А.С			Чат на основі веб-сервісу Схема структурна.	Літ. / Арк. 1 / Аркушів 1
Перевір.		Виноградов Ю				
Н.Контроль		Сімоненко В.				НТУУ КПІ ім. І. Сікорського Гр. ІО-61
Затверд.						



					ІАЛЦ.467100.04 Д1	
Змн.	Арк.	№ докум.	Підпис	Дата		
Розроб.		Коваленко А.С			Чат на основі веб-сервісу Схема принципова.	Літ. / Арк. 1 / Аркушів 1
Перевір.		Виноградов Ю				
Н.Контроль		Сімоненко В.				НТУУ КПІ ім. І. Сікорського Гр. ІО-61
Затверд.						



Змн.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.467100.04 Д1		
Розроб.		Коваленко А.С			Чат на основі веб-сервісу Схема функціональна	Літ.	Арк. 1 / Аркушів 1
Перевір.		Виноградов Ю					
Н.Контроль		Сімоненко В.				НТУУ КПІ ім. І. Сікорського Гр. ІО-61	
Затверд.							

ДОДАТОК Б

ЧАТ НА ОСНОВІ ВЕБ-СЕРВІСУ

ЛІСТИНГ ПРОГРАМИ

Аркушів 16

Київ – 2020

Client.ts

```
import * as websocketClient from "socket.io-client";
const clc = require("cli-color");
const Repl = require("repl");

const socketConnection = websocketClient(
  http://localhost:${process.argv[2] || "3333"},
  {
    transports: ["websocket"]
  }
);

socketConnection.on("connect", (err) => {
  console.log(clc.green("Client connected !!!"));

  initSocketEventListeners();
  initCommandLineInterface();
});

function initSocketEventListeners() {
  socketConnection.on("write", (data) =>
    console.log(`${clc.blue("Message")} from ${data.dialog} ==> ${data.msg}`)
  );
  socketConnection.on("memberAdded", (data) =>
    console.log(
      Member ${data.member} was ${clc.green("added")} to room ${data.dialog}
    )
  );
```

```
  socketConnection.on("memberRemoved", (data) =>
    console.log(
      Member ${data.member} was ${clc.green("removed")} from ${data.dialog}
    )
  );
  socketConnection.on("statusChange", (data) =>
    console.log(
      `User ${data.user} is ${
        data.online ? clc.green("online") : clc.green("offline")
      } now`
    )
  );
}

function initCommandLineInterface() {
  const repl = Repl.start({
    prompt: "> ",
    useColors: true,
    replMode: Repl.REPL_MODE_STRICT,
    ignoreUndefined: true
  });

  repl.context.auth = (ruid) => {
    socketConnection.emit("auth", ruid.toString(), (uid) => {
      console.log(`${clc.green("Authorized")} as ${uid}`);
    });
  };

  repl.context.logout = () => {
```

```
  socketConnection.emit("logout");
};

repl.context.subs = (unsub, sub) => {
  socketConnection.emit("updStatSubs", { unsub, sub });
};

repl.context.online = (users) => {
  socketConnection.emit("whoIsOnline", users, (res) =>
    console.log(`${clc.blue("Online users")}: ${res.join(", ")}`)
  );
};

repl.context.write = (dialog, msg) => {
  socketConnection.emit("write", {
    dialog: dialog.toString(),
    msg: msg.toString()
  });
};

repl.context.createChat = (dialog) => {
  socketConnection.emit("createChat", {
    dialog: dialog.toString()
  });
};

repl.context.join = (dialog, member) => {
  socketConnection.emit("addMember", {
    dialog: dialog.toString(),
```

```
      member: member.toString()
    });
  };

  repl.context.remove = (dialog, member) => {
    socketConnection.emit("removeMember", {
      dialog: dialog.toString(),
      member: member.toString()
    });
  };
}
```

message-controller.ts

```
import RedisController from "./redis-clients-controller";
import { ClientOpts } from "redis";
import * as msgpack from "notepack.io";

type EventCallback = (data: unknown, channel?: string) => void;

export default class MessageController {
  private readonly subscribers: RedisController;
  private readonly publishers: RedisController;
  private readonly clientsListenersList: {
    [index: string]: EventCallback[];
  }[] = [];

  constructor(opts: ClientOpts[]) {
    this.publishers = new RedisController(opts);
    this.subscribers = new RedisController(opts);
```

```
  this.initListeners(opts);
}

initListeners(opts: ClientOpts[]) {
  for (let index = 0; index < opts.length; index++) {
    const eventListeners = {};
    this.clientsListenersList.push(eventListeners);
    this.subscribers
      .getServerByIndex(index)
      .on("messageBuffer", (buff: Buffer, data: Buffer) => {
        const passedChannel = buff.toString();
        if (!eventListeners[passedChannel]) return;
        eventListeners[passedChannel].forEach((c) =>
          c(msgpack.decode(data), passedChannel)
        );
      });
  }
}

join(channel: string, callback: EventCallback): void {
  const clientIndex = this.subscribers.getIndexOfServer(channel);
  const eventListeners = this.clientsListenersList[clientIndex];
  if (!eventListeners[channel]) eventListeners[channel] = [];
  const callbacksList = eventListeners[channel];

  if (callbacksList.length == 0) {
    this.subscribers.getServerByIndex(clientIndex).subscribe(channel);
  }
```

```
  if (!!~callbacksList.indexOf(callback))
    throw new Error(`callback for "${channel}" already exists`);
  callbacksList.push(callback);
}

leave(channel: any, callback: EventCallback): void {
  const clientIndex = this.subscribers.getIndexOfServer(channel);
  const eventListeners = this.clientsListenersList[clientIndex];
  if (!eventListeners[channel])
    throw new Error(`Channel "${channel}" does not have any listeners`);
  const callbacksList = eventListeners[channel];
  const cb_index = callbacksList.indexOf(callback);
  if (!~cb_index)
    throw new Error(`Callback for "${channel}" does not exists`);

  callbacksList.splice(cb_index, 1);

  if (callbacksList.length > 0) return;

  this.subscribers.getServerByIndex(clientIndex).unsubscribe(channel);
  delete eventListeners[channel];
}

publish(channel: string, data: unknown): void {
  const clientIndex = this.subscribers.getIndexOfServer(channel);
  this.publishers
    .getServerByIndex(clientIndex)
    .publish(channel, msgpack.encode(data));
```

```
  }

  async getNumOfSubs(channels: string[]): Promise<{ [index: string]: number }>
{
    const sep: { [index: number]: string[] } = channels.reduce(
      (channel, key) => {
        const index = this.subscribers.getIndexOfServer(key);
        channel[index] = channel[index] || [];
        channel[index].push(key);
        return channel;
      },
      {}
    );
    return (
      await Promise.all<{ [index: string]: number }>(
        Object.keys(sep).map(
          (index) =>
            new Promise((res, rej) => {
              this.publishers
                .getServerByIndex(parseInt(index))
                .pubsub("NUMSUB", sep[index], (err, lock) => {
                  if (err) rej(err);
                  const subsc = <string[]>(<any>lock);
                  const cib = {};
                  while (subsc.length > 0) {
                    const l = subsc.shift();
                    const z = subsc.shift();
                    if (!l || !z) continue;
                    cib[l] = z;
```

```
            }
            res(cib);
          });
        })
      )
    )
  ).reduce((channel, key) => {
    return { ...channel, ...key };
  }, {});
 }
}
```

redis-clients-controller.ts

```
import { RedisClient, ClientOpts } from "redis";
import * as xxhash from "xxhash";

export default class RedisController {
 private readonly servers: RedisClient[] = [];

 constructor(options: ClientOpts[]) {
  options.forEach((opt) => this.servers.push(this._getRedisClient(opt)));
 }

 getServerByKey(key: string): RedisClient {
  return this.servers[this.getIndexOfServer(key)];
 }

 getIndexOfServer(key: string | number): number {
  if (typeof key == "number") key = key.toString();
```

```
    return Math.floor(
      xxhash
        .hash64(Buffer.from(key, "utf8"), 0x2b0352df, "buffer")
        .readUInt32BE() % this.servers.length
    );
  }

  getServerByIndex(index: number): RedisClient {
    return this.servers[index];
  }

  get length(): number {
    return this.servers.length;
  }

  _getRedisClient(options: ClientOpts): RedisClient {
    const client = new RedisClient(options);
    client.on("error", (err) => console.error(err));
    return client;
  }
}
```

index.ts

```
const clc = require("cli-color");
import RedisController from "./redis-clients-controller";
import MessageController from "./message-controller";

const msgController = new MessageController([
  {
```

```
    host: "message_bus_0",
    port: 6379
  },
  {
    host: "message_bus_1",
    port: 6379
  }
]);

class DialogsController {
  private readonly dialogsCache: RedisController;

  constructor() {
    this.dialogsCache = new RedisController([
      {
        host: "dialogs_cache",
        port: 6379
      }
    ]);
  }

  async get(dialogId: string): Promise<string[]> {
    return await new Promise<string[]>((resolve, reject) => {
      this.dialogsCache
        .getServerByKey(dialogId)
        .smembers(dialogId, (err, item) => (err ? reject(err) : resolve(item)));
    });
  }
```

```
  async getFor(userId: string, dialogId: string): Promise<string[]> {
    const members = await this.get(dialogId);
    if (!~members.indexOf(userId))
      throw new Error(`User "${userId}" forbidden to dialog "${dialogId}"`);
    return members;
  }

  add(dialogId, member): void {
    this.dialogsCache.getServerByKey(dialogId).sadd(dialogId, member);
  }

  remove(dialogId, member): void {
    this.dialogsCache.getServerByKey(dialogId).srem(dialogId, member);
  }
}

const dialogsController = new DialogsController();

interface Message {
  method: string;
  data: string;
}

import * as SocketIO from "socket.io";
const io = SocketIO(3000, {});

function toUserChan(uid: string): string {
  return u:${uid};
}
```

```
function toOnlineStatChan(uid: string): string {
  return s:${uid};
}

function fromUserChan(channel: string): string {
  return channel.slice(2);
}

io.on("connection", (sock) => {
  let UID = "";

  function receiveMessage(data: unknown): void {
    const msg: Message = <Message>data;
    sock.emit(msg.method, msg.data);
  }

  async function auth(uid, resp) {
    console.log(`User ${uid} authorized`);
    const uchan = toUserChan(uid);
    const subs = await msgController.getNumOfSubs([uchan]);
    if (!subs[uchan]) msgController.publish(toOnlineStatChan(uid), true);
    msgController.join(uchan, receiveMessage);
    resp(uid);
    UID = uid;
  }

  sock.on("auth", async (uid, resp) => {
    await auth(uid, resp);
```

```
});

async function logout() {
  if (!UID) return;
  const uchan = toUserChan(UID);
  msgController.leave(uchan, receiveMessage);
  const subs = await msgController.getNumOfSubs([uchan]);
  if (!subs[uchan]) msgController.publish(toOnlineStatChan(UID), false);
  UID = "";
}

sock.on("logout", async () => {
  await logout();
});

const statSubs = new Set();

function receiveStatus(data, channel): void {
  sock.emit("statusChange", {
    user: fromUserChan(channel),
    online: data
  });
}

sock.on("updStatSubs", async (changes) => {
  if (changes.unsub)
    changes.unsub.forEach((uid) => {
      const uchan = toOnlineStatChan(uid);
      if (!statSubs.has(uchan))
```

```
        throw new Error(`You already unsubscribed from ${uid}`);
      statSubs.delete(uchan);
      msgController.leave(uchan, receiveStatus);
    });
  if (changes.sub)
    changes.sub.forEach((uid) => {
      const uchan = toOnlineStatChan(uid);
      if (statSubs.has(uchan))
        throw new Error(`You already subscribed to ${uid}`);
      statSubs.add(uchan);
      msgController.join(uchan, receiveStatus);
    });
});

sock.on("createChat", async (data) => {
  dialogsController.add(data.dialog, UID);
  msgController.publish(toUserChan(UID), {
    method: "memberAdded",
    data: {
      dialog: data.dialog,
      member: UID
    }
  });
});

sock.on("write", async (data) => {
  const members = await dialogsController.getFor(UID, data.dialog);
  members.forEach((u) => {
    msgController.publish(toUserChan(u), {
```

```
        method: "write",
        data: data
      });
    });
  });

  sock.on("addMember", async (data) => {
    const members = await dialogsController.getFor(UID, data.dialog);
    dialogsController.add(data.dialog, data.member);
[...members, data.member].forEach((u) => {
      msgController.publish(toUserChan(u), {
        method: "memberAdded",
        data: data
      });
    });
  });

  sock.on("removeMember", async (data) => {
    const members = await dialogsController.getFor(UID, data.dialog);
    dialogsController.remove(data.dialog, data.member);
    members.forEach((u) => {
      msgController.publish(toUserChan(u), {
        method: "memberRemoved",
        data: data
      });
    });
  });

  sock.on("whoIsOnline", async (users: string[], resp) => {
```

```
    const subs = await msgController.getNumOfSubs(
      users.map((u) => toUserChan(u))
    );
    resp(Object.keys(subs).map((uc) => fromUserChan(uc)));
  });

  async function cleanup(): Promise<void> {
    statSubs.forEach((s) => msgController.leave(s, receiveStatus));
    await logout();
  }

  sock.on("disconnect", async (err) => {
    console.log(`Disconnecting ${err}`);
    await cleanup();
  });
});

process.on("unhandledRejection", (err) => {
  console.error(clc.red(`ERR: ${err}`));
});
```